

EXHIBIT 1

EXHIBIT 2

EXHIBIT 3



US005737547A

United States Patent [19]

Zuravleff et al.

[11] Patent Number: 5,737,547

[45] Date of Patent: Apr. 7, 1998

[54] **SYSTEM FOR PLACING ENTRIES OF AN OUTSTANDING PROCESSOR REQUEST INTO A FREE POOL AFTER THE REQUEST IS ACCEPTED BY A CORRESPONDING PERIPHERAL DEVICE**

[75] Inventors: William K. Zuravleff, Mountainview;
Mark Semmelmeier, Sunnyvale;
Timothy Robinson, Boulder Creek;
Scott Furman, Union City, all of Calif.

[73] Assignee: MicroUnity Systems Engineering, Inc., Sunnyvale, Calif.

[21] Appl. No.: 480,739

[22] Filed: Jun. 7, 1995

[51] Int. Cl.⁶ G06F 13/36; G06F 9/22

[52] U.S. Cl. 395/292; 395/307; 395/478;
395/497.01

[58] Field of Search 395/200.15, 825.
395/826, 860, 292, 307, 497.01, 497.02,
200.7, 478, 250

[56] **References Cited****U.S. PATENT DOCUMENTS**

3,654,621	4/1972	Bock et al.	340/172.5
4,473,880	9/1984	Budde et al.	364/200
4,502,115	2/1985	Eguchi	364/200
4,833,655	5/1989	Wolf et al.	365/221
5,043,981	8/1991	Firoozmand et al.	370/85.1
5,179,688	1/1993	Brown et al.	395/425
5,208,490	5/1993	Yetter	307/452
5,249,297	9/1993	Brockmann et al.	395/725
5,257,356	10/1993	Brockmann et al.	395/725
5,289,403	2/1994	Yetter	365/49
5,299,158	3/1994	Mason et al.	365/189.04
5,317,204	5/1994	Yetter	307/443
5,329,176	7/1994	Müller, Jr. et al.	307/443
5,343,096	8/1994	Heikes et al.	307/480
5,485,586	1/1996	Brush et al.	395/292
5,541,912	7/1996	Choudhury et al.	370/17
5,563,920	10/1996	Fimoff et al.	375/354

OTHER PUBLICATIONS

Eric DeLano et al., "A High Speed Superscaler PA-RISC Processor", IEEE, pp. 116-121, 1992.

Doug Hunt, "Advanced Performance Features of the 64-bit PA-8000", IEEE, pp. 123-128, 1995.

Jonathan Lotz et al., "A CMOS RISC CPU Designed for Sustained High Performance on Large Applications", IEEE Journal of Solid-State Circuits, vol. 25, pp. 1190-1198, Oct. 1990.

William Jaffe et al., "A 200MFLOP Precision Architecture Processor", Hewlett-Packard, pp. 1.2.1-1.2.13.

Primary Examiner—Thomas C. Lee

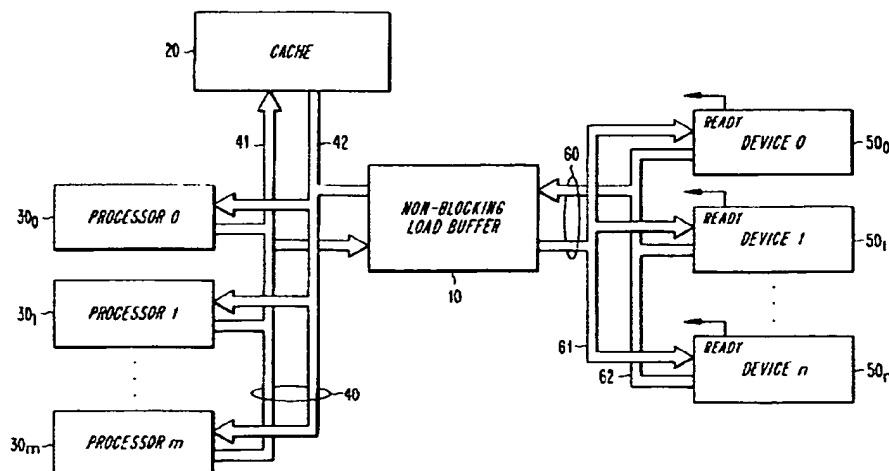
Assistant Examiner—Rehana Perveen

Attorney, Agent, or Firm—Burns, Doane, Swecker & Mathis, L.L.P.

[57] **ABSTRACT**

A non-blocking load buffer is provided for use in a high-speed microprocessor and memory system. The non-blocking load buffer interfaces a high-speed processor/cache bus, which connects a processor and a cache to the non-blocking load buffer, with a lower speed peripheral bus, which connects to peripheral devices. The non-blocking load buffer allows data to be retrieved from relatively low bandwidth peripheral devices directly from programmed I/O of the processor at the maximum rate of the peripherals so that the data may be processed and stored without unnecessarily idling the processor. I/O requests from several processors within a multiprocessor may simultaneously be buffered so that a plurality of non-blocking loads may be processed during the latency period of the device. As a result, a continuous maximum throughput from multiple I/O devices by the programmed I/O of the processor is achieved and the time required for completing tasks and processing data may be reduced. Also, a multiple priority non-blocking load buffer is provided for serving a multiprocessor running real-time processes of varying deadlines by prioritization-based scheduling of memory and peripheral accesses.

27 Claims, 10 Drawing Sheets

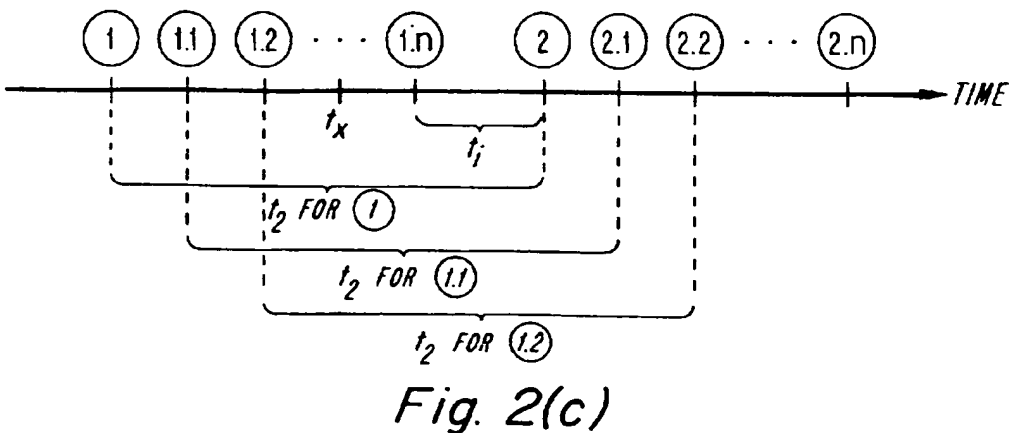
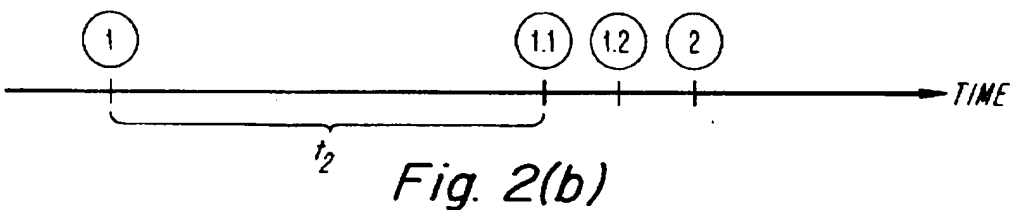
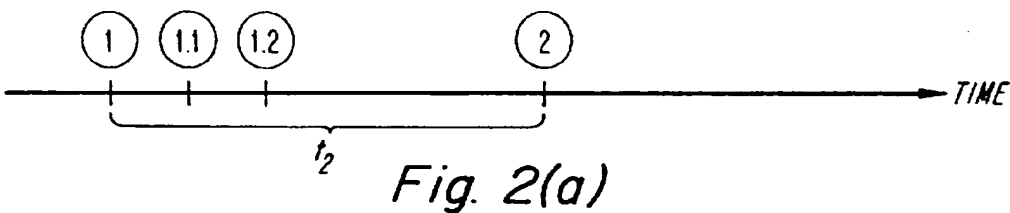
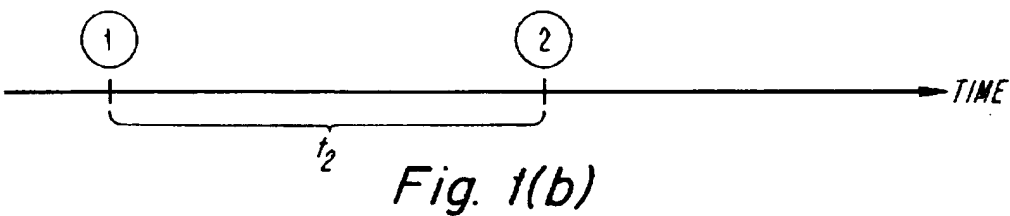
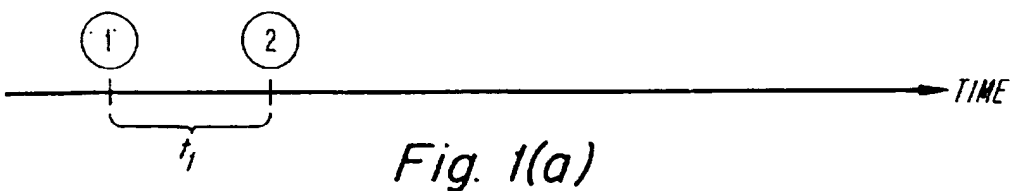


U.S. Patent

Apr. 7, 1998

Sheet 1 of 10

5,737,547



U.S. Patent

Apr. 7, 1998

Sheet 2 of 10

5,737,547

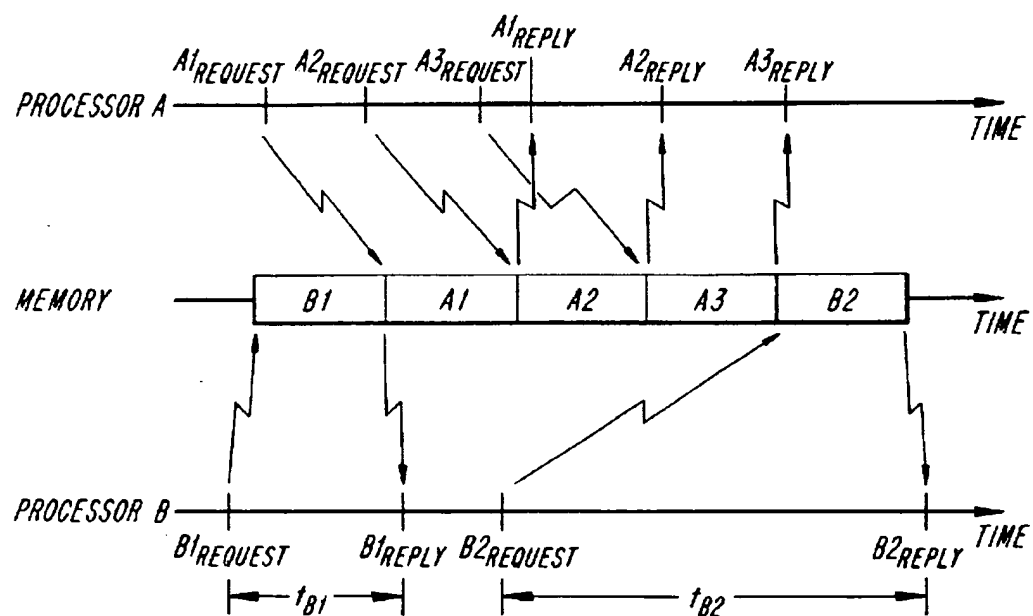


Fig. 3(a)

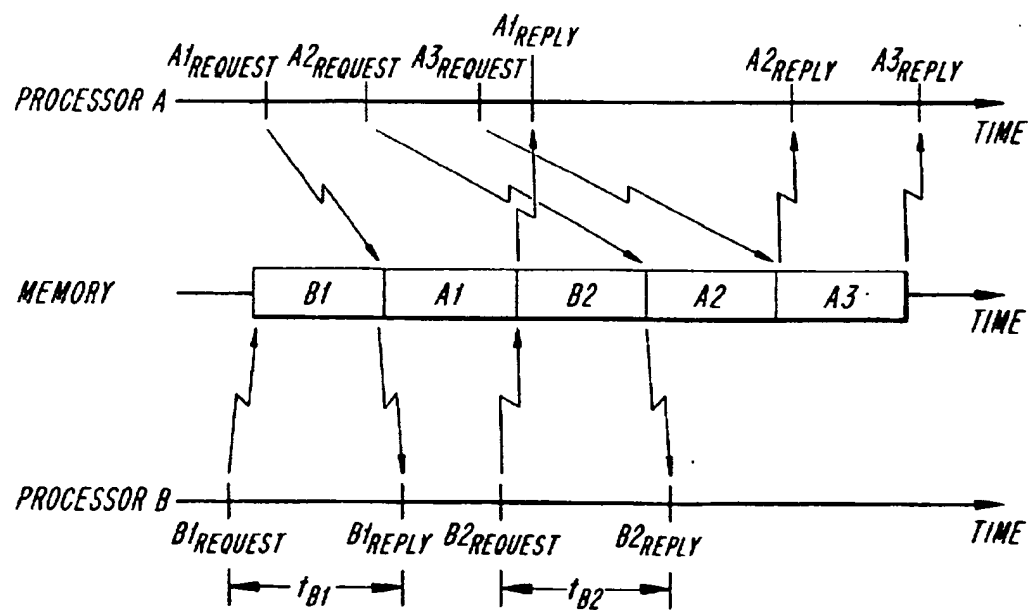


Fig. 3(b)

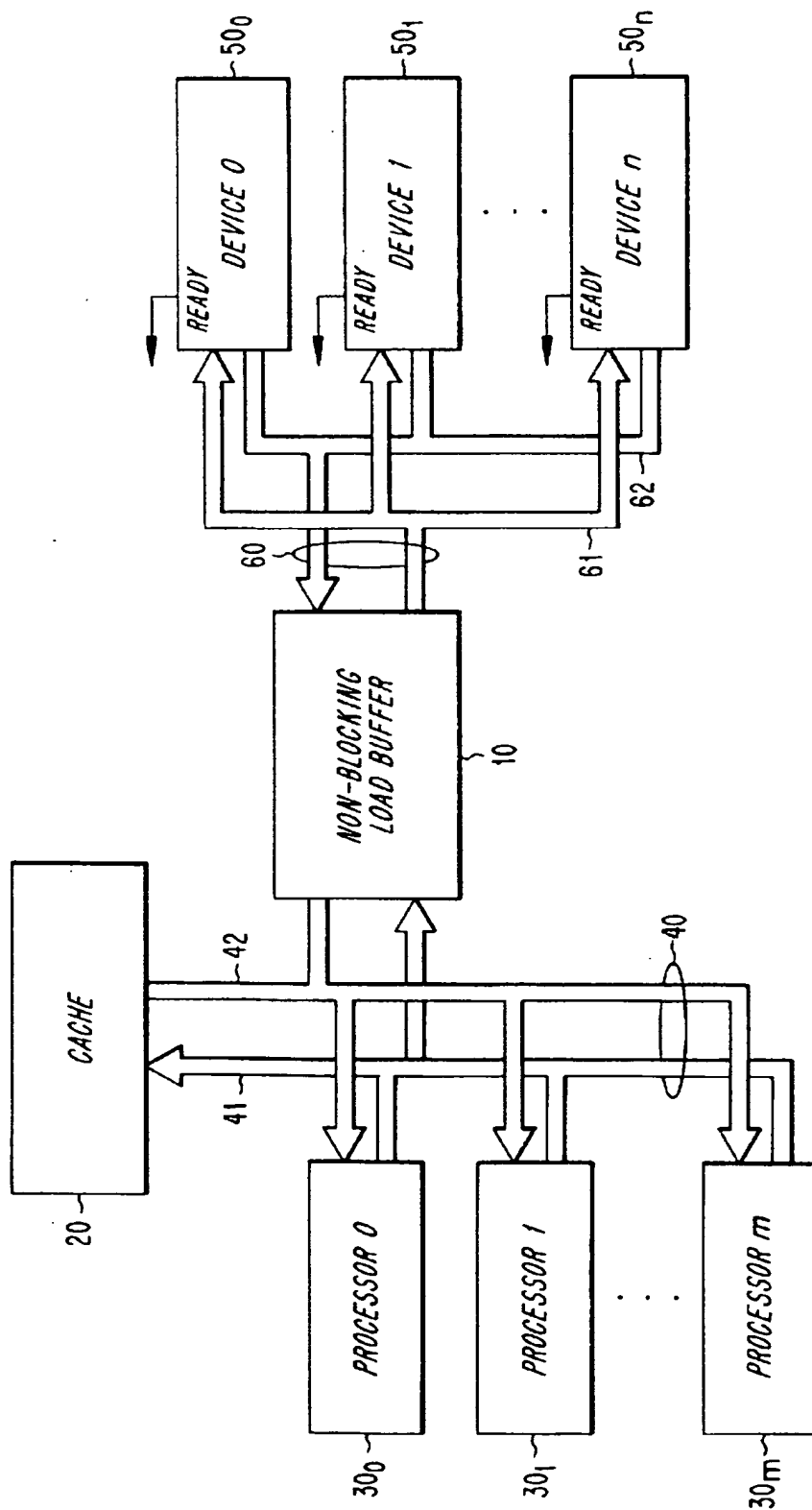
U.S. Patent

Apr. 7, 1998

Sheet 3 of 10

5,737,547

Fig. 4



U.S. Patent

Apr. 7, 1998

Sheet 4 of 10

5,737,547

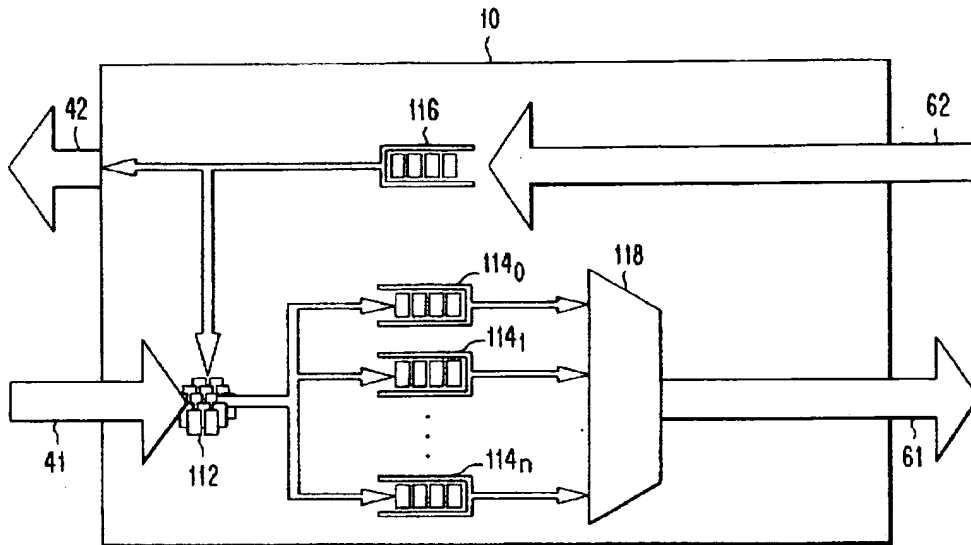


Fig. 5

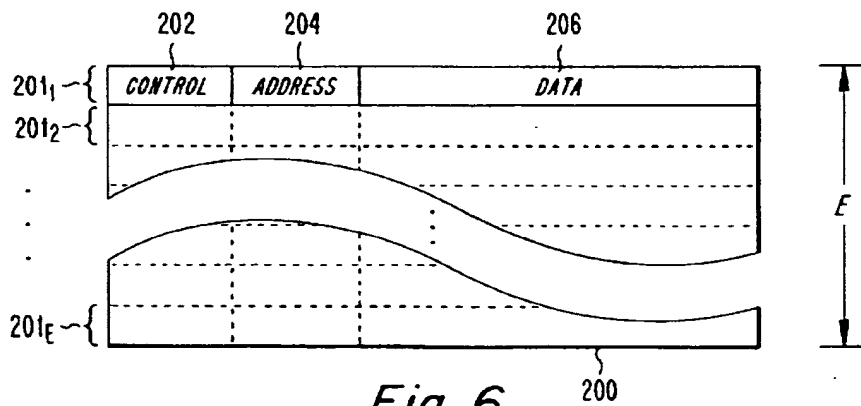


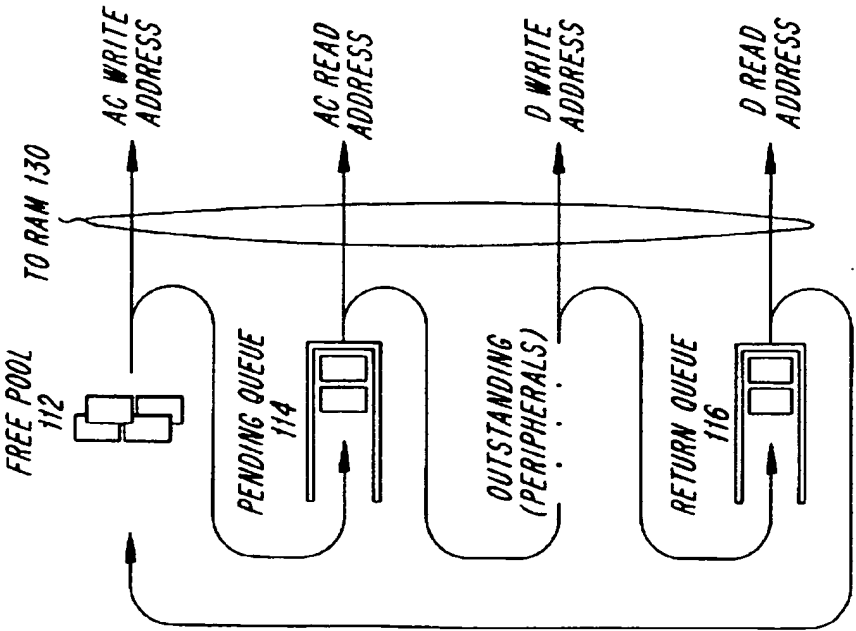
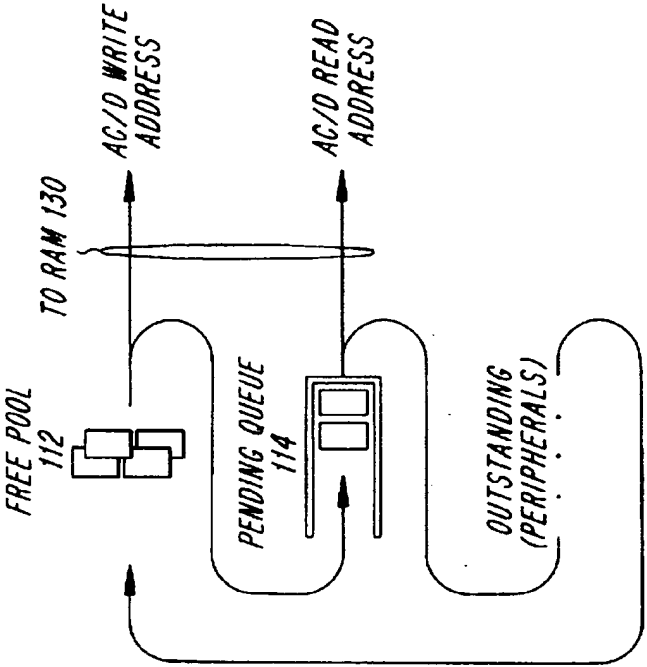
Fig. 6

U.S. Patent

Apr. 7, 1998

Sheet 5 of 10

5,737,547



U.S. Patent

Apr. 7, 1998

Sheet 6 of 10

5,737,547

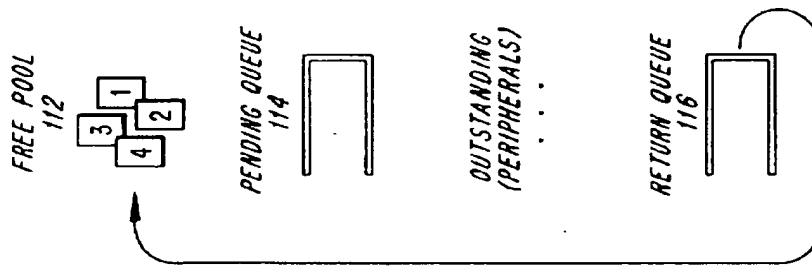


Fig. 7(c)

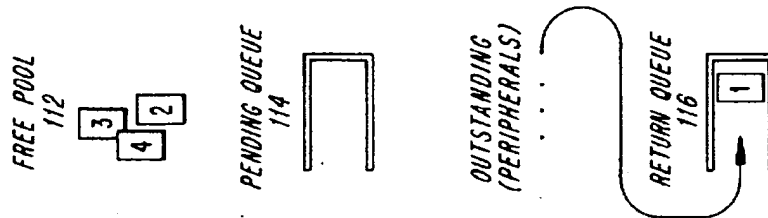


Fig. 7(d)

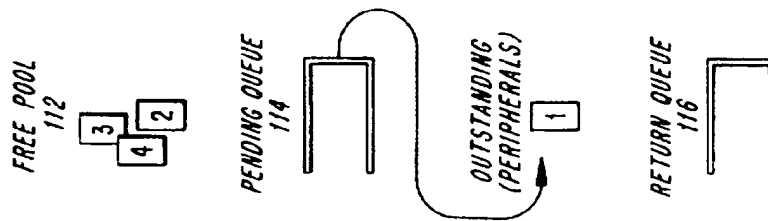


Fig. 7(e)

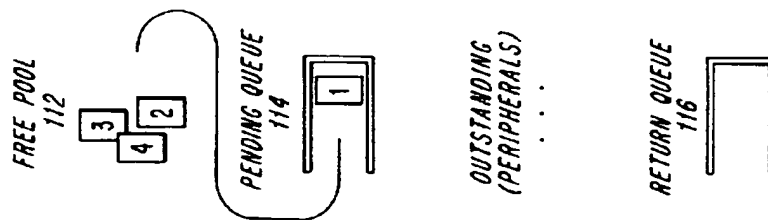


Fig. 7(f)

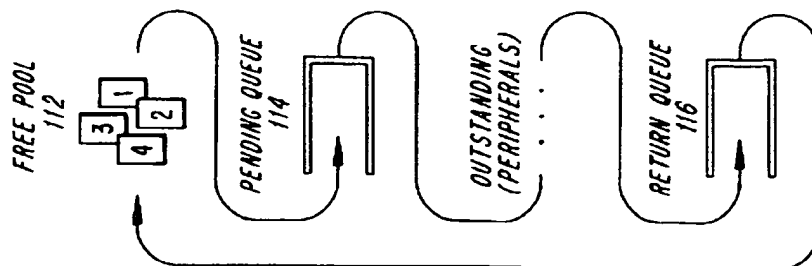


Fig. 7(g)

U.S. Patent

Apr. 7, 1998

Sheet 7 of 10

5,737,547

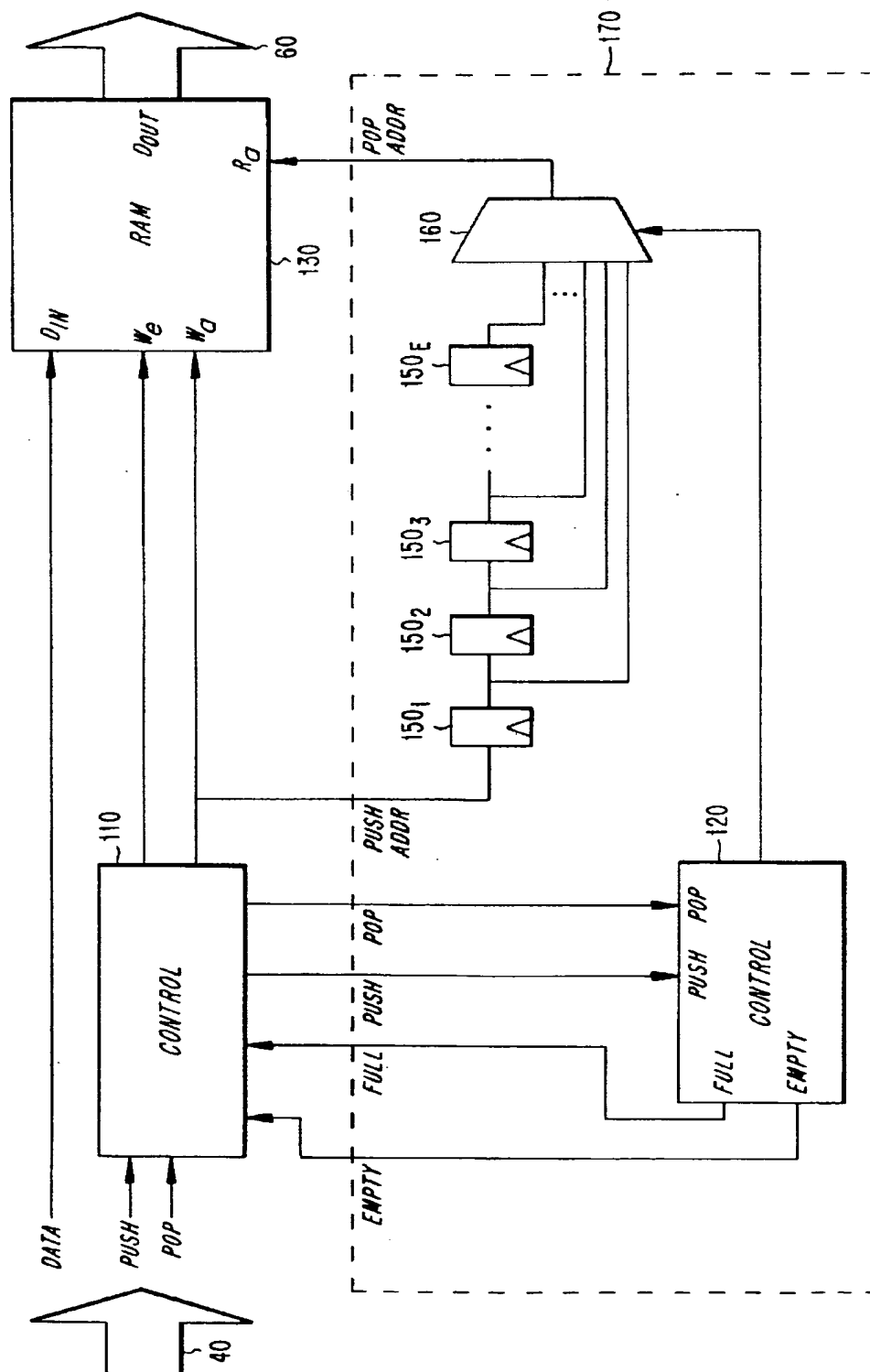


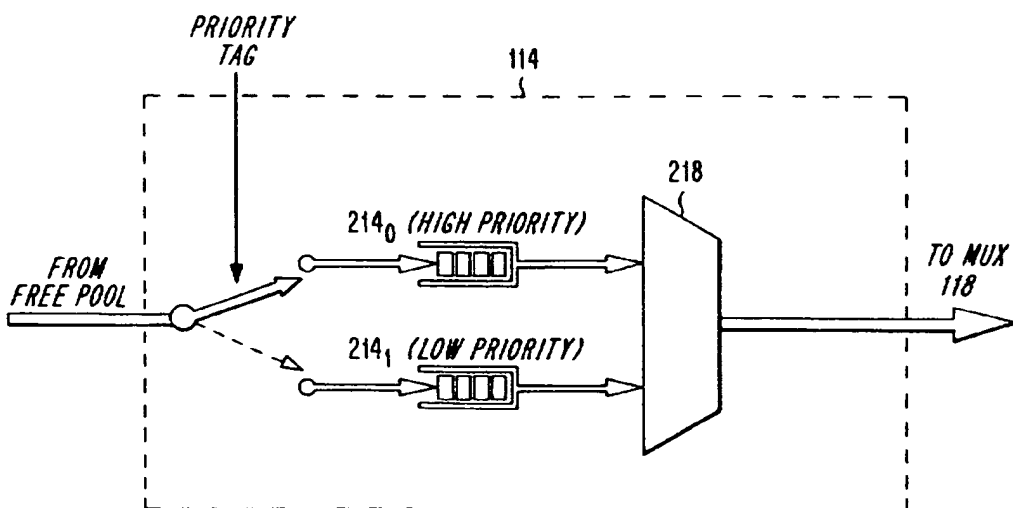
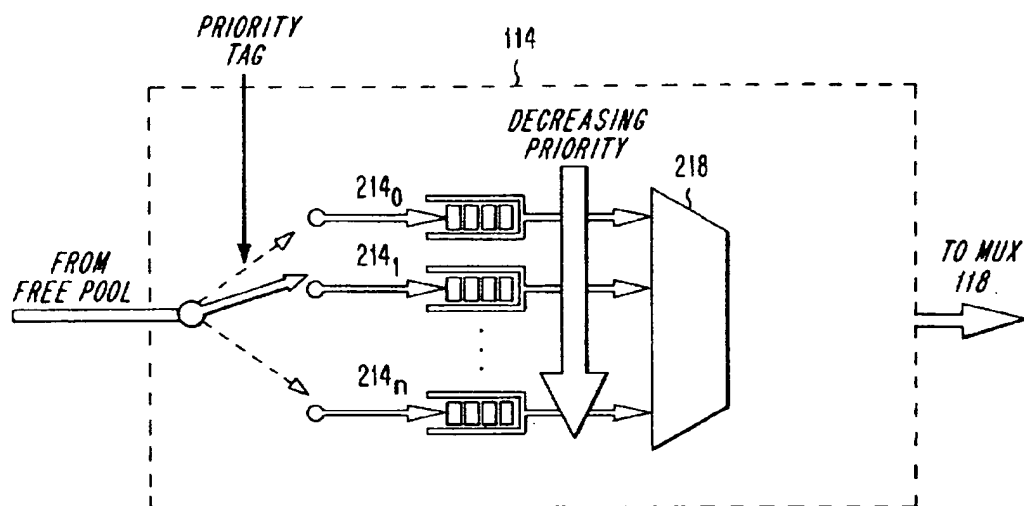
Fig. 8

U.S. Patent

Apr. 7, 1998

Sheet 8 of 10

5,737,547



U.S. Patent

Apr. 7, 1998

Sheet 9 of 10

5,737,547

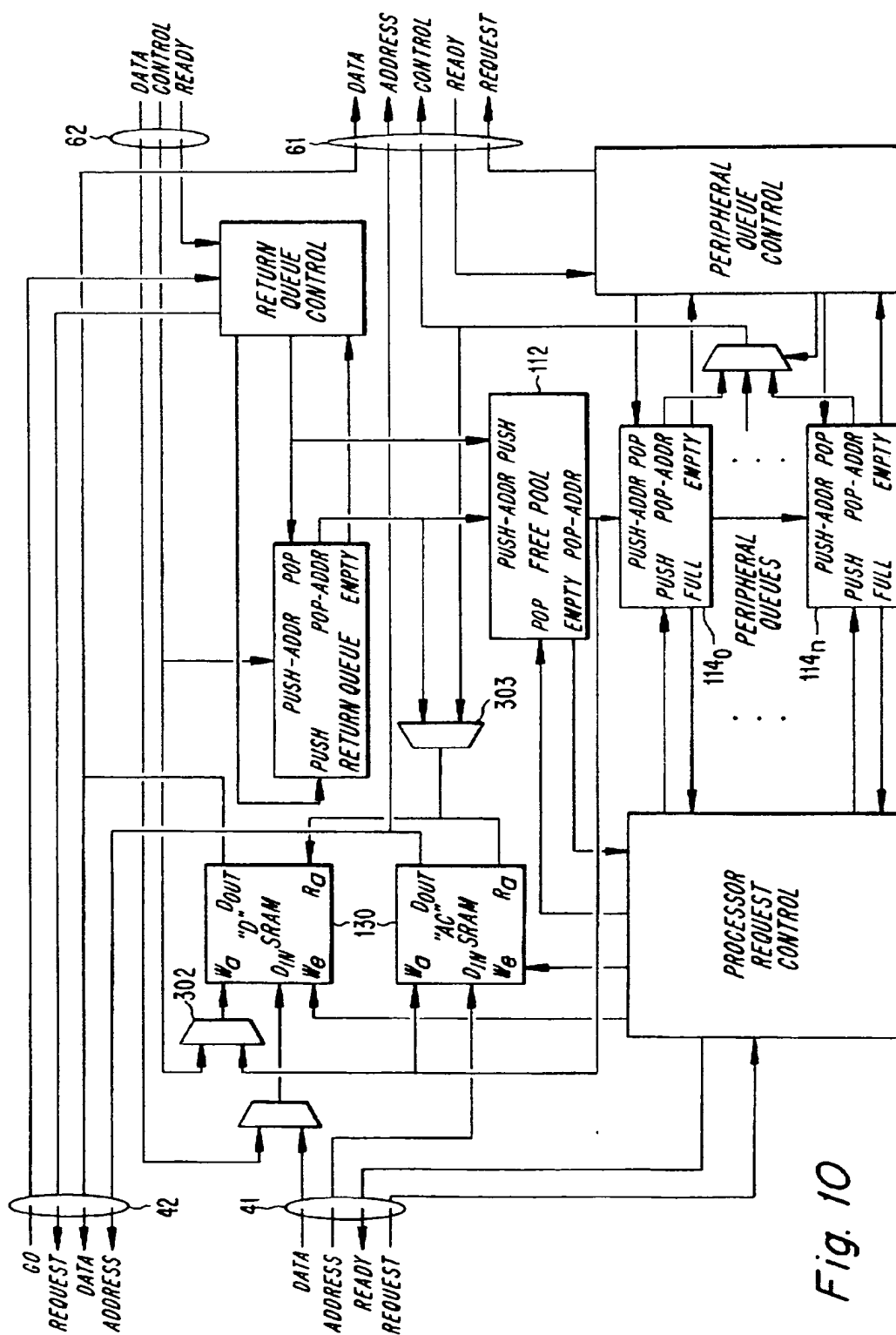


Fig. 10

U.S. Patent

Apr. 7, 1998

Sheet 10 of 10

5,737,547

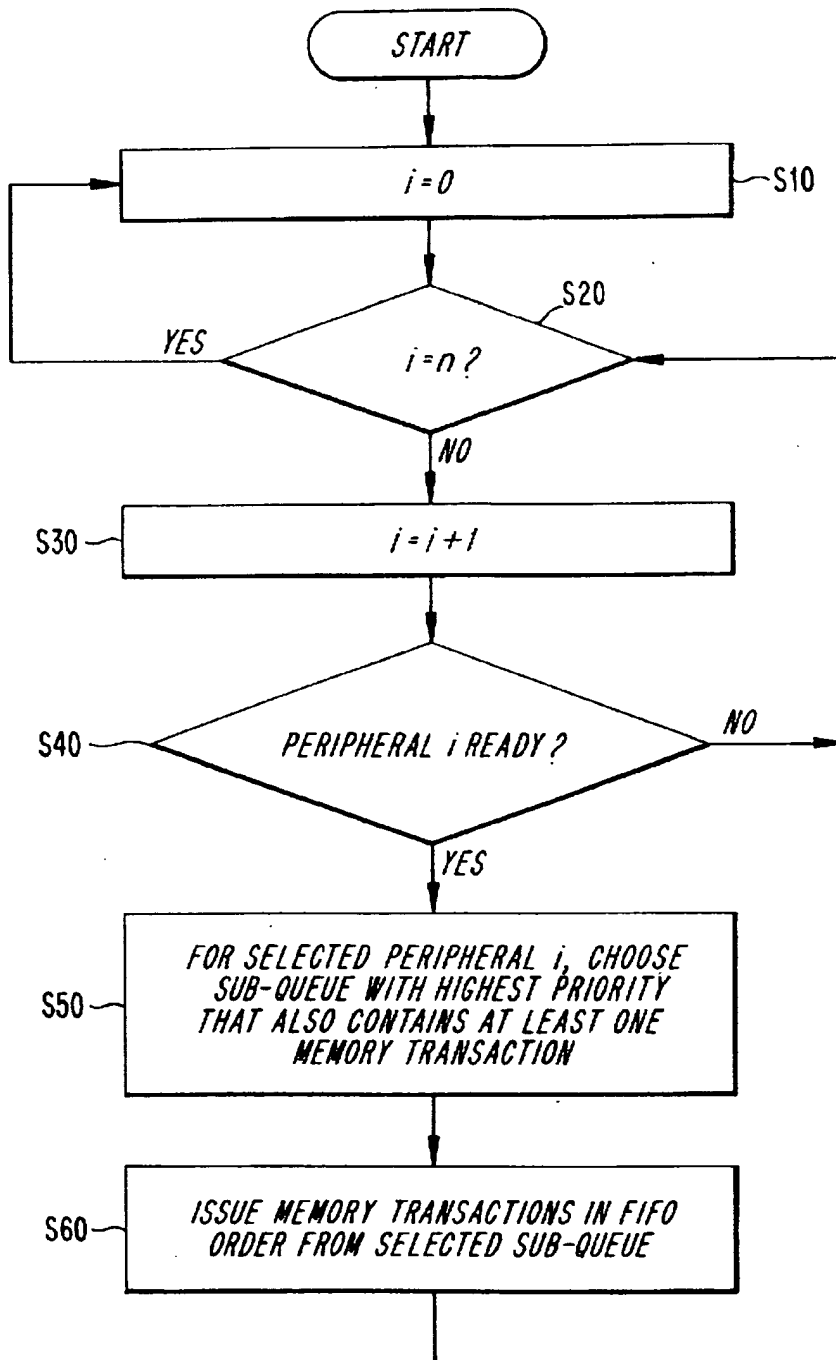


Fig. 11

5,737,547

1

**SYSTEM FOR PLACING ENTRIES OF AN
OUTSTANDING PROCESSOR REQUEST
INTO A FREE POOL AFTER THE REQUEST
IS ACCEPTED BY A CORRESPONDING
PERIPHERAL DEVICE**

**CROSS REFERENCE TO RELATED
APPLICATION**

This application is related to a co-pending application filed on Jun. 7, 1995 having a Ser. No. 08/480,738 (now pending).

FIELD OF THE INVENTION

The present invention relates to interfacing between a microprocessor, peripherals and/or memories. More particularly, the present invention relates to a data processing system for interfacing a high-speed data bus, which is connected to one or more processors and a cache, to a second, possibly lower speed peripheral bus, which is connected to a plurality of peripherals and memories, by a non-blocking load buffer so that a continuous maximum throughput from multiple I/O devices is provided via programmed I/O (memory-mapped I/O) of the processors and a plurality of non-blocking loads may be simultaneously performed. Also, the present invention is directed to a multiple-priority non-blocking load buffer which serves a multiprocessor for running real-time processes of varying deadlines by priority-based non-FIFO scheduling of memory and peripheral accesses.

BACKGROUND OF THE INVENTION

Conventionally, microprocessor and memory system applications retrieve data from relatively low bandwidth I/O devices, process the data by a processor and then store the processed data to the low bandwidth I/O devices. In typical microprocessor memory system applications, a processor and a cache are directly coupled by a bus to a plurality of peripherals such as I/O devices and memory devices. However, the processor may be unnecessarily idled due to the latency between the I/O devices and the processor, thus causing the processor to stall and, as a result, excessive time is required to complete tasks.

In known processing systems, when an operation is performed on one of the peripherals such as a memory device, the time between performing such an operation and subsequent operations is dependent upon the latency period of the memory device. Thereby, the processor will be stalled during the entire duration of the memory transaction. One solution for improving the processing speed of known processing systems is to perform additional operations during the time of the latency period as long as there is no load-use dependency upon the additional operations. For example, if data is loaded into a first register by a first operation (1), where the first operation (1) corresponds to:

load

$r1 \leftarrow r2$ (1)

and the first register is added to another operation by a second operation (2) where the second operation (2) corresponds to:

add

$r3 \leftarrow r1 + r4$ (2)

the operation (2) is load-use dependent and the operation (2) must wait for the latency period before being performed.

2

FIGS. 1(a) and 1(b) illustrate a load-use dependent operation where the time for initiating the operation (2) must wait until the operation (1) is completed. Operation (2) is dependent upon the short latency period t_1 in FIG. 1(a) corresponding to a fast memory and the long latency period t_2 in FIG. 1(b) corresponds to a slower memory.

A non-blocking cache and a non-blocking processor are known where a load operation is performed and additional operations other than loads may be subsequently performed during the latency period as long as the operation is not dependent upon the initial load operation. FIGS. 2(a) and 2(b) illustrate such operations. In operation (1), the first register is loaded. Next, operations (1.1) and (1.2) are to be executed. As long as operations (1.1) and (1.2) are not load dependent on another load, these operations may be performed during the latency period t_2 as illustrated in FIG. 2(a). However, if operation (1.1) is either load-dependent or a pending load, operations (1.1) and (1.2) must wait until the latency period t_2 ends before being performed.

Also known is a Stall-On-Use (Hit Under Miss) operation for achieving cache miss optimizations as described in "A 200 MFLOP Precision Architecture Processor" at Hot Chips IV, 1993, William Jaffe et al. In this Hit Under Miss operation, when one miss is outstanding only certain other types of instructions may be executed before the system stalls. For example, during the handling of a load miss, execution proceeds until the target register is needed as an operand for another instruction or until another load miss occurs. However, this system is not capable of handling two misses being outstanding at the same time. For a store miss, execution proceeds until a load or sub-word store occurs to the missing line.

This Hit Under Miss feature can improve the runtime performance of general-purpose computing applications. Examples of programs that benefit from the Hit Under Miss feature are SPEC benchmarks, SPICE circuit simulators and gcc C compilers. However, the Hit Under Miss feature does not sufficiently meet the high I/O bandwidth requirements for digital signal processing applications such as digital video, audio and RF processing.

Known microprocessor and memory system applications use real-time processes which are programs having deadlines corresponding to times where data processing must be completed. For example, an audio waveform device driver process must supply audio samples at regular intervals to the output buffers of the audio device. When the driver software is late in delivering data for an audio waveform device driver, the generated audio may be interrupted by objectionable noises due to an output buffer underflowing.

In order to analyze whether or not a real-time process can meet its deadlines under all conditions requires predictability of the worst-case performance of the real-time processing program. However, the sensitivity of the real-time processing program to its input data or its environment makes it impractical in many cases to exhaustively check the behavior of the process under all conditions. Therefore, the programmer must rely on some combination of analysis and empirical tests to verify that the real-time process will complete in the requisite time. The goals of real-time processing tend to be incompatible with computing platforms that have memory or peripheral systems in which the latency of the transactions is unpredictable because an analysis of whether the real-time deadlines can be met may not be possible or worst-case assumptions of memory performance are required. For example, performance estimates can be made by assuming that every memory transaction takes the maximum possible time. However, such an

5,737,547

3

assumption may be so pessimistic that any useful estimate for the upper bound on the execution time of a real-time task cannot be made. Furthermore, even if the estimates are only slightly pessimistic, overly conservative decisions will be made for the hardware performance requirements so that a system results that is more expensive than necessary.

Also, it is especially difficult to reliably predict real-time processing performance on known multiprocessors because the memory and peripherals are not typically multi-ported. Therefore simultaneous access by two or more processors to the same memory device must be serialized. Even if a device is capable of handling multiple transactions in parallel, the bus shared by all of the processors may still serialize the transactions to some degree.

If memory requests are handled in a FIFO manner by a known multiprocessor, a memory transaction which arrives slightly later than another memory transaction may take a much longer amount of time to complete since the later arriving memory requests must wait until the earlier memory request is serviced. Due to this sensitivity, very small changes in the memory access patterns of a program can cause large changes in its performance. This situation grows worse as more processors share the same memory. For example, if ten processors attempt to access the same remote memory location simultaneously, the spread in memory latency among the processors might be 10:1 because as many as nine of these memory transactions may be buffered for later handling. In general, it is not possible to predict which of these processors will suffer the higher latencies and which of these processors will receive fast replies to their memory accesses. Very small changes to a program or its input data may cause the program to exhibit slight operation differences which perturb the timing of the memory transactions.

Furthermore, types of memory which exhibit locality effects may exacerbate the above-described situation. For example, accesses to DRAMs are approximately two times faster if executed in page mode. To use page mode, a recent access must have been made to an address in the same memory segment (page). One of the most common access patterns is sequential accesses to consecutive locations in memory. These memory patterns tend to achieve high page locality, thus achieving high throughput and low latency. Known programs which attempt to take advantage of the benefits of page mode may be thwarted when a second program executing on another processor is allowed to interpose memory transactions on a different memory page. For instance, if ten processors, each with its own sequential memory access pattern, attempt to access the same DRAM bank simultaneously and each of the accesses is to a different memory page, the spread and memory latencies between the fastest and slowest responses might be more than 25:1.

The present invention is directed to allowing a high rate of transfer to memory and I/O devices for tasks which have real-time requirements. The present invention is also directed to allowing the system to buffer I/O requests from several processors within a multiprocessor at once with a non-blocking load buffer. Furthermore, the present invention is directed to extending the basic non-blocking load buffer to service a data processing system running real-time processes of varying deadlines by using scheduling of memory and peripheral accesses which is not strictly FIFO scheduling.

SUMMARY OF THE INVENTION

An object of the present invention is to reduce the effect of memory latency by overlapping a plurality of non-blocking loads during the latency period and to achieve a

4

similar reduction on the effect of latency for non-blocking stores without requiring the additional temporary storage memory of a separate store buffer.

Another object of the present invention is to provide a non-blocking load buffer which buffers I/O requests from one or more processors within a multiprocessor so that a plurality of I/O memory transactions, such as loads and stores, may be simultaneously performed and the high I/O bandwidth requirements for digital signal processing applications may be met.

A still further object of the present invention is to provide a multiple-priority non-blocking load buffer for serving a multiprocessor running real-time processes of varying deadlines by using a priority-based method to schedule memory and peripheral accesses.

The objects of the present invention are fulfilled by providing a data processing system comprising a first data bus for transferring data requests at a first speed, a second bus for transmitting I/O data of a second speed, and a non-blocking load buffer connected to the first and second data buses for holding the data requests and the I/O data so that a plurality of loads and stores may be performed simultaneously. It is possible for the speed of the second bus to be slower than the first speed of the first bus. As a result, data may be retrieved from relatively low bandwidth I/O devices and the retrieved data may be processed and stored to low bandwidth I/O devices without idling the system unnecessarily.

In a further embodiment for a data processing system of the present invention, the first data bus is connected to a processor and the second data bus is connected to a plurality of peripherals and memories. The data processing system for this embodiment reduces the effect of latency of the peripherals and memories so that the application code, which uses programmed I/O, may meet its real-time constraints. Furthermore, the code may have reduced instruction scheduling constraints and a near maximum throughput may be achieved from the I/O devices of varying speeds.

Another embodiment of the present invention is fulfilled by providing a non-blocking load buffer comprising a memory array for temporarily storing data, and a control block for simultaneously performing a plurality of data loads and stores. The non-blocking load buffer for this embodiment allows I/O requests from several processors to be performed at once.

Further scope of applicability of the present invention will become apparent from the detailed description given hereafter. However, it should be understood that the detailed description and specific examples, while indicating preferred embodiments of the invention, are given by way of illustration only, since various changes and modifications within the spirit and scope of the invention will become apparent to those skilled in the art from this detailed description.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will become more fully understood from the detailed description given hereinbelow and the accompanying drawings which are given by way of illustration only, and thus are not limitative of the present invention and wherein:

FIGS. 1(a) and 1(b) illustrate time dependency in a conventional load use operation;

FIGS. 2(a) and 2(b) illustrate time dependency in a known Stall-On-Use operation;

5,737,547

5

FIG. 2(c) illustrates time dependency in a non-blocking load buffer for an embodiment of the present invention;

FIG. 3(a) illustrates unpredictable memory latency in a known multiprocessor system;

FIG. 3(b) illustrates memory latency in a multiprocessor system having a multiple-priority version of the non-blocking load buffer;

FIG. 4 illustrates a block diagram for the data processing system according to one embodiment of the present invention;

FIG. 5 is a schematic illustration of the non-blocking load buffer for an embodiment of the present invention;

FIG. 6 illustrates an example of the contents for the entries in the memory of the data processing system for an embodiment of the present invention;

FIGS. 7(a) and 7(b) illustrate examples of the possible states for a non-blocking load through the memory of the data processing system for an embodiment of the present invention;

FIGS. 7(c), 7(d), 7(e), 7(f), and 7(g) illustrate the progress of addresses through queues of the data processing system in an embodiment of the present invention;

FIG. 8 illustrates the circuitry used for the non-blocking load buffer in an embodiment of the present invention;

FIGS. 9(a) and 9(b) illustrate parallel pending queues which allow prioritization of data in the data processing system for an embodiment of the present invention;

FIG. 10 illustrates a detailed block diagram of an embodiment of the non-blocking load buffer; and

FIG. 11 illustrates a flow chart of the control of the pending queue in an embodiment of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The first embodiment of the present invention will be discussed with reference to FIG. 4. The data processing system includes a non-blocking load buffer 10, a cache 20, one or more processors 30_{0,m}, . . . 30_n, a processor/cache bus 40 for connecting the non-blocking load buffer 10, the cache 20 and the processors 30_{0,m}, a plurality of peripheral devices 50₀, 50₁, . . . 50_n, and a peripheral bus 60, which includes an output bus 61 and an input bus 62, for connecting the non-blocking load buffer 10 with the plurality of peripheral devices 50_{0,m}. Some or all of the peripheral devices 50_{0,m} may be memories, such as DRAM for example, which serve as a backing store for the cache 20 or as memory which the processors 30_{0,m} access directly through the non-blocking load buffer 10, thus bypassing the cache 20.

The processor/cache bus 40, which includes an input bus 41 and an output bus 42, transfers data from the cache 20 to registers of the processors 30_{0,m} as loads via the output bus 42 or alternatively, from the registers of the processors 30_{0,m} to the cache 20 as stores via the input bus 41. All I/O operations, which are programmed I/O, and any memory operations, even those which bypass the cache 20, are transmitted over the processor/cache bus 40 as well. For such operations the source or sink of the data is one of the peripheral devices 50_{0,m}, such as a DRAM, for example.

The bandwidth of the peripheral bus 60 should be chosen to meet the peak input or output bandwidth which is to be expected by the peripheral devices 50_{0,m}. The bandwidth of the non-blocking load buffer 10 should therefore be chosen to at least equal the sum of the bandwidth of the processor/cache bus 40 and the bandwidth of the peripheral bus 60.

6

The processors 30_{0,m} may be designed with dependency-checking logic so that they will only stall when data requested by a previously executed load instruction has not yet arrived and that data is required as a source operand in a subsequent instruction. FIG. 2(c) illustrates the time dependency for the non-blocking load buffer 10 in an embodiment of the present invention. In FIG. 2(c), load instructions are executed at operations (1), (1.1), (1.2), . . . (1.n). In this example, operations (1), (1.1), (1.2), . . . (1.n) correspond to:

```

load
    r1←[r2]                                (1);
load
    r5←[r2+24]                             (1.1);
load
    r6←[r2+32]                             (1.2);
. . . function
    z1                                      (1.n)

```

In this example, operation (2) executes an instruction dependent on operation (1), operation (2.1) executes an instruction dependent on operation (1.1), operation (2.2) executes an instruction dependent on operation (1.2), . . . operation (2.n) executes an instruction dependent on operation (1.n). For instance, operations (2.1), (2.2), . . . (2.n) correspond to:

```

add
    r3←r1+r4                              (2);
add
    r7←r5+r4                              (2.1);
add
    r8←r6+r4                              (2.2);
. . . function
    z2                                      (2.n)

```

n may be any positive integer. Operations (1.1), (1.2), . . . (1.n) occur as fast as possible and the time between these operations is dependent upon the processor cycle for each operation. As shown in FIG. 2(c), operation (2) waits only for a part of the latency period t_2 between operations (1) and (2), which is represented by t_1 , since the part of the latency period t_2 between operations (1) and (2) is overlapped with the operations (1.1), (1.2), . . . (1.n). Thereafter, operations (2.1) and (2.2) occur as fast as possible after operation (2) with the time limitation being the processor cycle. The operations (2.1) and (2.2) do not wait for the full latency period t_2 of operation (1.1) since the latency period t_2 is partially overlapped with operations (1.1), (1.2), . . . (1.n) and the latency period t_2 for operation (1). For values of n sufficiently large, t_1 may effectively be reduced to zero and the processor will not suffer any stalls due to memory or I/O latency. In this embodiment, there may be more than one outstanding load to slower memories or peripherals, as shown at instant t_1 in FIG. 2(c) for

5,737,547

7

example, and requests may thereby be pipelined to provide continuous maximum throughput from the plurality of peripheral devices 50_{0..n} from programmed I/O of the processor.

The non-blocking load buffer 10 may accept both load and store requests from the processors 30_{0..m} to the peripheral devices 50_{0..n}. The non-blocking load buffer 10 may accept requests at the maximum rate issued by a processor for a period of time that is limited by the amount of intrinsic storage in the non-blocking load buffer 10. After accepting requests, the non-blocking load buffer 10 then forwards the requests to the peripheral devices 50_{0..n} at a rate which the peripheral devices 50_{0..n} may process them. However, the non-blocking load buffer 10 does not have to wait until all of the requests are accepted.

FIG. 5 illustrates internal architecture for the non-blocking load buffer in an embodiment of the present invention. The internal architecture of the non-blocking load buffer 10 includes a plurality of variable-length pending queues 114₀, 114₁, . . . 114_n, which contain entries that represent the state of requests made by the processors 30_{0..m} to the peripheral devices 50_{0..n}. The pending queues 114_{0..n} correspond to each of the peripheral devices 50_{0..n} and are used to hold the address and control information for a load or a store before being transmitted to the corresponding peripheral device. If an entry in the pending queues 114_{0..n} contains a store transaction, then that entry will also contain the data to be written to the corresponding peripheral device. Another component of the non-blocking load buffer 10 is a variable-length return queue 116 used to buffer the data returned by any of the peripheral devices 50_{0..n} in response to a load until the data can be transmitted to the requesting processor. In addition, there is a free pool 112 that contains any unused request entries. Once a load request is sent from one of the pending queues 114_{0..n} to its corresponding peripheral device, the load request is marked as "outstanding" until a return data value is sent by the peripheral and enqueued in the return queue 116. At any time, a given request entry is either outstanding, in one of the pending queues 114_{0..n}, in the return queue 116, or is not in use and is therefore located in the free pool 112.

FIG. 6 illustrates an example of a memory structure 200 for storing the information present in each entry of the non-blocking load buffer 10. Entries may be stored in the non-blocking load buffer 10 and each entry holds one of a plurality of requests 201₁, 201₂, . . . 201_z which can either be a load or a store to any of the peripheral devices 50_{0..n}. Each request 201 includes a control information 202, an address 204 and data 206. However, for a load operation, the data 206 is empty until data returns from the peripheral devices 50_{0..n}. The control information 202 describes parameters such as whether the transaction is a load or a store, the size of data transferred, and a coherency flag that indicates whether all previous transactions must have been satisfied before the new transaction is permitted to be forwarded to the addressed peripheral device. The coherency flag enforces sequential consistency, which is occasionally a requirement for some I/O or memory operations. The data stored in the entries 201 need not be transferred among the queues of the non-blocking load buffer 10. Rather pointers to individual entries within the memory structure 200 may be exchanged between the pending queues 114_{0..n}, the return queue 116 and the free pool 112 for example.

FIGS. 7(a) and 7(b) illustrate examples of the operation of the non-blocking load buffer 10 which will be input to a peripheral device such as a RAM. Initially, all entries of the non-blocking load buffer 10 are unused and free. With

8

reference to FIG. 7(a), when one of the processors 30_{0..m} issues a non-blocking load, a free entry is chosen and the address of the load is stored in this entry. Additionally, space for the returning data is allocated. This entry now enters the pending queue, one of the queues 114_{0..n}, corresponding to the addressed peripheral device. When the peripheral device is ready, the pending entry is delivered over the peripheral bus 60 and becomes an outstanding entry. When the outstanding entry returns from the peripheral device via the peripheral bus 60, the accompanying requested load data is written to the part of the entry that had been allocated for the returning data and the entry is placed in the return queue 116. The data is buffered until the one processor is ready to receive the data over the processor/cache bus 40 (i.e., until the one processor is not using any required cache or register file write ports). Finally, the data is read out and returned to the one processor over the processor/cache bus 40 and the entry again becomes free. In addition, a peripheral device may accept multiple requests before returning a response.

FIG. 7(b) illustrates the progress of an entry during a store operation. The store operation is similar to the load operation but differs because data does not need to be returned to the processors 30_{0..m} and this return step is, therefore, skipped. Once a store is accepted by the non-blocking load buffer 10, the store is treated by the issuing processor as if the data has already been transferred to the target peripheral device, one of the peripheral devices 50_{0..n}.

The non-blocking load buffer 10 performs similarly to a rate-matching FIFO but differs in several ways. For instance, one difference is that although memory transactions forwarded by the non-blocking load buffer 10 are issued to any particular peripheral device 50_{0..n} in the same order that they were requested by any of the processors 30_{0..m}, the transactions are not necessarily FIFO among different peripheral devices. This is similar to having a separate rate-matching FIFO for outputting to each of the peripheral devices 50_{0..n}, except that the storage memory for all the FIFOs is shared and is thereby more efficient. Also, the peripheral devices 50_{0..n} are not required to return the results of a load request to the non-blocking load buffer 10 in the same order that the requests were transmitted.

The non-blocking load buffer 10 is capable of keeping all of the peripheral devices 50_{0..n} busy because an independent pending queue exists for each of the peripheral devices. A separate flow control signal from each of the peripheral devices indicates that this corresponding peripheral device is ready. Accordingly, a request for a fast peripheral device does not wait behind a request for a slow peripheral device.

FIGS. 7(c), 7(d), 7(e), 7(f) and 7(g) illustrate examples of a non-blocking load progressing through the queues. After the queues are reset, entries 1, 2, 3 and 4 are in the free pool as shown in FIGS. 7(c). A non-blocking load is pending after the entry 1 is written into one of the pending queues as shown in FIG. 7(d). Next, FIG. 7(e) illustrates that entry 1 becomes outstanding after the pending load is accepted by its corresponding peripheral device. Subsequently, the load is returned from the peripheral device and entry 1 is now in the returned queue as illustrated in FIG. 7(f), and awaits transmission to the requesting processor (or to the cache if the access was part of a cache fill request). Finally, in FIG. 7(g), entry 1 is released to the free pool and, once again, all entries are free.

FIG. 8 illustrates the circuitry used for the non-blocking load buffer in an embodiment of the present invention which implements variable depth FIFOs. This circuitry manipulates pointers (i.e. RAM addresses for example) to entries in a RAM 130 instead of storing the addresses of only the head

5,737,547

9

and tail of a contiguous list of entries in the RAM 130. Furthermore, the circuitry does not require the pointers to be sequential. Therefore, records need not be contiguous and the RAM 130 can contain several interleaved data structures allowing more flexible allocation and de-allocation of entries within a FIFO so that it is possible to use the same storage RAM 130 for multiple FIFOs. The entire circuit in FIG. 8 represents a large FIFO of an unspecified data width which is controlled by a smaller FIFO represented by block 170 having pointer widths smaller than the records. In FIG. 8, a first control circuit 110, a second control circuit 120, the RAM 130 having separate read and write ports, a plurality of shift registers 150₁, 150₂, 150₃, . . . 150_p, corresponding to each entry in the RAM 130 and a multiplexer 160 are illustrated. The control circuits 110 and 120 provide pointers to the entries of the RAM 130 rather than the entries themselves being queued. Block 170 is essentially a FIFO of pointers. A pointer may be enqueued in the FIFO of pointers in block 170 by asserting the "push" control line active and placing the pointer (SRAM 130 address) on the "push-addr" input lines to block 170. Similarly, a pointer value may be dequeued by asserting the "pop" control line on block 170 with the pointer value appearing on the "pop-addr" bus. Block 170 has a "full" output that when active indicates that no more pointers may be enqueued because all the registers 150_{1-p} are occupied and the queue is at its maximum depth. p. Block 170 also has an "empty" output that indicates when no pointers remain enqueued.

In a preferred embodiment, the RAM 130 is divided into two arrays, an address control array AC and a data array D. The address control array AC stores the control information 202 and the address 204 for each entry and the data array D stores the data 206 for each entry. Dividing the non-blocking load buffer into these two arrays AC and D realizes a more efficient chip layout than using a single array for both purposes.

FIG. 10 illustrates the entire structure for a non-blocking load buffer in an embodiment of the present invention. Each of the pending queues 114_{0-n} and the return queue 116 may be composed of copies of block 170 illustrated in FIG. 8 for example. The queue entries are stored in the AC and D arrays 501 and 502, with pointers to the array locations in the flip-flops that compose the pending queues 114_{0-n}, the return queue 116 and the free pool 112. For the embodiment illustrated in FIG. 10, the write ports to the AC and D arrays are time-division multiplexed using multiplexers 301 and 302 to allow multiple concurrent accesses from the processors 30_{0-m}, which issue load and store requests over bus 41, and the peripheral devices 50_{0-n}, which return requested load data over bus 62. Similarly, the read ports are time-division multiplexed by multiplexer 303 between returning load data going back to the processors 30_{0-m} via bus 42 and issuing of load and store requests to the peripherals 50_{0-n} via bus 61. A processor request control unit 510 accepts the load and store requests from the processors 30_{0-m}, a return queue control unit 520 controls the data returned by the peripheral devices 50_{0-n} to the return queue 116 and a pending queue control unit 530 controls the selection of the pending queue 114_{0-n}.

When either a non-blocking load buffer or a conventional single FIFO load buffering operation is used, the uncertainty and memory latency is further compounded because individual processors may transmit more than one memory request at a time as illustrated in FIG. 3(a) for example. In FIG. 3(a), processor A issues three non-blocking loads to a memory device after a first non-blocking load is issued by processor B but before a second non-blocking load is issued

10

by processor B to the same memory device. The first memory transaction issued by processor B, labeled B1, will be executed immediately. However, the latency of the second transaction, labeled B2, by processor B, t_{B2} , will be considerably longer than the latency of the first transaction, t_{B1} , since the second transaction B2 will not be handled by the memory system until all of the requests A1, A2, and A3 by processor A have been processed.

One consideration for overcoming all of these unpredictable memory latency effects is to have the programmer carefully design the times at which each real-time process attempts to access the memory which effectively has software serialize access to the memory banks instead of hardware. For instance, two processors may alternate memory accesses at regular intervals to match the throughput capabilities of the memory system. However, such programming places a heavy burden on both the program and the programmer because the programmer must be aware of the derailed temporal behavior of the memory access patterns by the program. The behavior of the programs is usually very complex and may change based on its input data, which is not always known. Therefore, it is impractical for the programmer to know the detailed temporal behavior of the memory access patterns by the program.

Also, because the combinations of processors that are making the requests are even more difficult to predict and the combinations of programs running on the different processors might not always be the same on each occasion or even known, the derailed temporal behavior of the memory access patterns by the program may not be known. For example, two processes that run two different programs on two different processors may have been written by different programmers and the source code may not even be available for either or both programs. As a result, this programming approach for overcoming the unpredictable memory latency defeats the purpose of using a non-blocking load buffer, which was designed to simplify the burden on the programmer and the compiler by relaxing the scheduling constraints under which memory operations can be issued while still permitting efficient utilization of the processor.

FIG. 9(a) illustrates another embodiment of the present invention for a multiple-priority version of the non-blocking load buffer. In this embodiment, the non-blocking load buffer is a variation of the basic non-blocking load buffer where multiple pending sub-queues 214₀, 214₁, . . . 214_p exist as a component of each pending queue 114_{0-n}. The outputs of the sub-queues 214_{0-p} are then input to the sub-multiplexers 218 associated with one of the peripheral devices. Each output of the sub-multiplexers 218 are then input to the main multiplexer 118. Each of these pending sub-queues is assigned a unique priority level. In the simplest implementation of this multiple-priority version of the non-blocking load buffer, illustrated in FIG. 9(b), there are two pending sub-queues 214₀ and 214₁ for a peripheral device with sub-queue 214₀ being assigned a high priority and sub-queue 214₁ assigned a low priority. The multiple priority non-blocking load buffer issues memory or peripheral transactions in a highest-priority-first manner. In other words, no transaction will be issued from a pending sub-queue for a given peripheral unless all of the higher priority sub-queues for that same peripheral are empty. For each peripheral device, requests are issued from the pending sub-queue in a FIFO manner for memory transactions of the same priority.

The computational processes associated with the memory transactions that take place at each of these relative priority levels execute on the processors 30_{0-m}. The priority levels

5,737,547

11

are assigned by the processors based on the scheduling parameters of all the processes in the system. The priority level of the process is designed to reflect the relative importance of the memory accesses with respect to allowing all of the processes to meet their deadlines if possible. The priority levels may also be applicable in a uni-processor environment. For example, interrupt handlers may be provided to achieve low latency by using higher priority memory transactions than the transaction being interrupted. The priority level may be identified by adding a priority tag to the memory transaction. This priority tag is used to channel the memory transaction into the pending sub-queue with the matching priority level, thus the selection of the appropriate destination pending sub-queue for a non-blocking memory transaction is a function of both the address and the priority level of the access. The priority tag may be stored with the other control information 202 in the queue entry 201 corresponding to a given non-blocking memory access.

FIG. 11 illustrates a flow chart for the functions performed by the pending queue control unit in an embodiment of the present invention for the multiple-priority version of the non-blocking load buffer. At step S10, the counter i for a pending queue is initialized to zero. At step S20, the counter is compared to the number n of pending queues 114. If the counter is not equal to the total number of pending queues, the counter is incremented at step S30 and a determination is made at step S40 of whether the peripheral corresponding to the counter is ready. If the peripheral device is determined not to be ready at step S40, the process returns to step S20. However, if the peripheral corresponding to the counter is ready at step S40, a pending queue is selected at step S50 with the highest priority that also contains at least one memory transaction. At step S60, memory transactions are issued in a FIFO order from the selected sub-queue and the process returns to step S20. If the counter is equal to the number of pending queues at S20, the process has been completed for all the pending queues and step S10 is returned to where the counter is initialized.

FIG. 3(b) illustrates memory latency for the multiple-priority non-blocking load buffer. In this example, processor B's memory transactions are assigned a higher priority than processor A's memory transactions. Therefore, transaction B2 is delivered to the memory before transactions A2 and A3 even though the request to begin transaction B2 arrived at the non-blocking load buffer after requests A2 and A3. As a result, the latency for transaction B2, t_{B2} , is less in FIG. 3(b) than t_{B2} in FIG. 3(a), which illustrates a non-blocking load buffer that does not offer the benefit of multiple-priority scheduling. Using the multiple-priority version of the non-blocking load buffer, Processor B spends less time stalled waiting for transaction B2 to complete as illustrated by the comparison in FIGS. 3(a) and 3(b).

These priority levels may be heuristic in nature. For example, if using earliest deadline first (EDF) scheduling, a process should not be assigned a lower priority than the priority of any process which has a more distant deadline. In general, this priority level is not necessarily fixed for each process and the priority level may vary over time as the demands of the real-time process or of other processes change. As another example of a priority selection mechanism, if the load-use distance is known for a load instruction (as computed by a compiler), it can be used to set a priority for each individual memory access instruction. (Higher load-use distances result in lower priorities.) In general, non real-time processes (the processes having no deadlines) are typically given the lowest priority.

12

Process priority may also be used to arbitrate for the limited resources within the non-blocking load buffer itself. For example, in one embodiment of the non-blocking load buffer, limited data storage memory is shared among all peripheral devices and processors. When the total number of slots among all of the non-blocking load buffer queues exceeds the number of non-blocking memory entries, it is possible for a process at a low priority to prevent a higher priority process from completely utilizing one or more of its pending sub-queues by using up these entries. The use of priority in allocating non-blocking memory entries can be used to eliminate this effect. For example, the maximum number of outstanding non-blocking memory transactions may be specified for each of the available priorities.

Once an appropriate priority has been determined for a process, the priority of its memory transactions might be specified to the non-blocking load buffer by employing any of several techniques. For example, the priority of any given memory transaction might be determined by an operand to the instruction that performs the memory access or the priority can be associated with certain bit combinations in either the virtual address, the physical address of the memory or the physical address of the peripheral device accessed. Alternatively, the processors might be designed with programmable registers that set the priority of all memory accesses for each processor. Yet another possible technique is to store the memory access priority in the page table of a virtual memory system and thus make the memory access priority a characteristic of specific memory pages. When these various techniques are combined with conventional memory management and processor design techniques, memory priorities can be treated as privileged resources. As a result, the operating system reserves the highest priority levels for its own real-time tasks and therefore the user level programs are extremely limited in their ability to disrupt important system real-time tasks.

To maintain memory coherency in a multiple-priority non-blocking load buffer, which is not always necessary, no load or store may be issued to any of the peripheral devices while a load or store to the same address is outstanding or while an earlier load or store to the same address is pending. The memory coherency may be maintained by time stamps combined with address comparison. However, memory coherency may also be maintained more simply by ensuring that requests of different priorities are sent to non-overlapping address segments within the same peripheral device.

In the normal operation of programmed I/O activity, the processors 30_{0,m} do not need to exactly schedule loads to achieve maximum throughput from the peripheral devices but can instead burst out requests to the limits of the non-blocking load storage and, if properly programmed, perform useful work while waiting for data to return. The non-blocking load buffer allows application code to use programmed I/O (memory mapped I/O) for achieving near maximum throughput from the I/O devices of varying speeds which reduces the effective latency of the I/O peripheral devices and relaxes the scheduling constraints on the programmed I/O. The non-blocking load buffer functions to rate-match the requests from the processor to the peripheral devices and back and to act upon the priority of requests from the processor to allow high priority requests to go ahead of low priority requests already buffered.

The non-blocking load buffer uses queues of pointers to centralize storage to increase storage density, parallel queues to implement requests of different priority and memory segment descriptors to determine priority. Accordingly, I/O

5,737,547

13

requests to a fast device need not wait behind requests to a slow device and requests from several processes of a signal processor running multiple processes are buffered so that the processor is not unnecessarily idled and the time to complete tasks is reduced. Because the multiple priority non-blocking load buffer has multiple pending sub-queues for each of the peripherals, a processor used in combination with this multiple priority non-blocking load buffer is able to run real-time processes of varying deadlines by use of non-FIFO scheduling of memory and peripheral accesses. Also, the multiple priority non-blocking load buffer simplifies the burden on the programmer and the compiler by relaxing the scheduling constraints under which memory operations can be issued while still permitting efficient utilization of the processor.

The invention being thus described, it will be obvious that the same may be varied in many ways. Such variations are not to be regarded as a departure from the spirit and scope of the invention, and all such modifications as would be obvious to one skilled in the art are intended to be included within the scope of the following claims.

What is claimed is:

1. In a data processing system including one or more requesting processors that generate processor requests directed to one or more peripheral devices, a non-blocking load buffer, comprising:

a plurality of variable depth pending queues corresponding to each one of the plurality of peripheral devices for queuing entries of processor requests;

a variable length return queuing unit for queuing data returned from said peripheral devices in response to an outstanding processor request; and

a free pool of entries for placing entries of the outstanding processor request, after the outstanding processor request is accepted by a corresponding peripheral device.

2. A non-blocking load buffer according to claim 1, wherein said variable length return queuing unit comprises one variable length return queue.

3. A non-blocking load buffer according to claim 1, wherein said variable length return queuing unit comprises a plurality of variable length return queues.

4. A non-blocking load buffer according to claim 3, wherein each of said plurality of variable length return queues corresponds to each of said requesting processors.

5. A non-blocking load buffer according to claim 3, wherein each of said plurality of variable length return queues corresponds to each of said peripheral devices.

6. A non-blocking load buffer according to claim 3, wherein each of said plurality of variable length return queues corresponds to a unique priority level.

7. A non-blocking load buffer according to claim 1, wherein each of said pending queues include a plurality of sub-queues, wherein each of said sub-queues is assigned a unique priority level.

8. A non-blocking load buffer according to claim 7, wherein said processor requests include an address and a priority tag, said address directs said processor requests to a corresponding one of said pending queues and said priority tag channels said processor requests to a corresponding one of said sub-queues within the one said pending queue.

9. A non-blocking load buffer according to claim 8, wherein each of said pending queues comprises a priority controller for issuing the processor requests from said sub-queues in a highest priority first manner.

14

10. A multiple priority non-blocking load buffer comprising:

a variable depth pending queue for queuing entries of memory or I/O requests generated by a processor to peripheral devices, said pending queue including a plurality of sub-queues with each sub-queue having a unique priority level assigned thereto;

a variable length return queuing unit for queuing data returned from said peripheral devices in response to an outstanding memory or I/O request; and

a free pool of entries for placing entries of the outstanding I/O request, after the outstanding I/O request is accepted.

11. A multiple priority non-blocking load buffer according to claim 10, wherein said variable length return queuing unit comprises one variable length return queue.

12. A multiple priority non-blocking load buffer according to claim 10, wherein said variable length return queuing unit comprises a plurality of variable length return queues.

13. A multiple priority non-blocking load buffer according to claim 12, wherein each of said plurality of variable length return queues corresponds to each of said peripheral devices.

14. A multiple priority non-blocking load buffer according to claim 12, wherein each of said plurality of variable length return queues corresponds to a unique priority level.

15. A multiple priority non-blocking load buffer according to claim 10, wherein said memory or I/O requests include a priority tag, said priority tag channels said memory or I/O requests to a corresponding one of said sub-queues.

16. A multiple priority non-blocking load buffer according to claim 15, wherein said pending queue comprises a priority controller for issuing said memory or I/O requests from said sub-queues in a highest priority first manner.

17. A non-blocking load buffer according to claim 7, wherein a maximum number of outstanding processor requests is specified for said unique priority level in each of said sub-queues which prevents entries of processor requests having low priority levels from using one of said sub-queues before entries of processor requests having higher priority levels.

18. A non-blocking load buffer according to claim 1, wherein priorities corresponding to the entries of processor requests are determined by logical memory addresses, control bits derived from a memory management page table, control bits derived from segmentation entries, virtual addresses of a memory management system, programmable registers which set priorities for each processor, instructions, or instruction operand.

19. A multiple priority non-blocking load buffer according to claim 10, wherein a maximum number of outstanding memory or I/O requests is specified for said unique priority level in each of said sub-queues which prevents entries of memory or I/O requests having low priority levels from using one of said sub-queues before entries of memory or I/O requests having higher priority levels.

20. A multiple priority non-blocking load buffer according to claim 10, wherein priorities corresponding to entries of memory or requests are determined by logical memory addresses, control bits derived from a memory management page table, control bits derived from segmentation entries, virtual addresses of a memory management system, programmable registers which set priorities for each processor, instructions, or instruction operand.

21. A non-blocking load buffer according to claim 1, wherein the variable depth pending queues queue the entries of processor requests using the free pool of entries.

5,737,547

15

22. The data processing system according to claim 1, wherein the non-blocking processor requests are generated by running real-time processes.

23. The data processing system according to claim 1, wherein the processor requests are prioritized in the pending queues such that the higher priority processor requests are processed before the lower priority processor requests.

24. The data processing system according to claim 23, wherein the pending queues are prioritized such that the higher priority pending queues have a higher number of maximum entries than the lower priority pending queues.

25. The data processing system according to claim 1, wherein the one or more requesting processors generate the processor requests over a first bus, and wherein the periph-

16

eral devices accept the processor requests over a second bus that has a different bus bandwidth from the first bus.

26. The data processing system according to claim 25, wherein the entries include pointers that point to memory locations on a shared memory device.

27. The data processing system according to claim 26, wherein the processor requests include control information, addresses and data, and wherein the shared memory device is partitioned for storing the address and control information in a first memory array and for storing the data in a second separate memory array.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE
Certificate

Patent No. 5,737,547

Patented: April 7, 1998

On petition requesting issuance of a certificate for correction of inventorship pursuant to 35 U.S.C. 256, it has been found that the above-identified patent, through error and without deceptive intent, improperly sets forth the inventorship.

Accordingly, it is hereby certified that the correct inventorship of this patent is: William K. Zuravleff, Mountainview, CA; Mark Semmelmeier, Sunnyvale, CA; Timothy Robinson, Boulder Creek, CA; Scott Furman, Union City, CA; Craig Hansen, Los Altos, CA.

Signed and Sealed this Nineteenth Day of September, 2000.

THOMAS C. LEE
Art Unit 2782

EXHIBIT 4

Internet.com

Developer

News

Small Business

Personal Tech

Events

Jobs

Partners

Solutions

Time

Logos

Register

BUSINESS SAVINGS ACCOUNT

5.00% APY

- HIGH INTEREST
- NO FEES
- NO MINIMUMS

ING DIRECT

Save Your

MEMBER **FDIC**

Internet.com

You are in the Small Business Computing Channel

View Sites

Program: Palm Device Loaner Program. Borrow Palm devices and use them to develop, troubleshoot, test, port, and applications that leverage Palm technology.

Internet.com (Webopedia) The #1 online encyclopedia dedicated to computer technology

Search All Sites

Enter a word for a definition...

...or choose a computer category.

ASIC

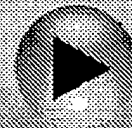
Last modified: Thursday, January 28, 2004

Pronounced *ay-sik*, and short for *Application-Specific Integrated Circuit*, a chip designed for a particular application (as opposed to the integrated circuits that control functions such as RAM in a PC). ASICs are built by connecting existing circuit building blocks in new ways. Since the building blocks already exist in a library, it is much easier to produce a new ASIC than to design a new chip from scratch.

ASICs are commonly used in automotive computers to control the functions of the

internet.com

VIDEO



Manage Your Windows Server With a Web Browser
Networking tip: Using the Web Administration feature in Windows Server 2003 systems can provide an easy way to perform many common management functions right from your Web browser.

Access Free Developer Tools

Download these IBM resources today!

e-Kit: IBM Rational Systems Development Solution

With systems teams under so much pressure to develop products faster, reduce production costs, and react to changing business needs quickly, communication and collaboration seem to get lost. Now, there's a way to improve product quality and communication.

Download: IBM Rational Rose Technical Developer V7.0

MENU

[Home](#)
[Term of the Day](#)
[New Terms](#)
[Pronunciation](#)
[New Links](#)
[Quick Reference](#)
[Did You Know?](#)
[Categories](#)
[Tech Support](#)
[Technology Jobs](#)
[About Us](#)
[Link to Us](#)
[Advertising](#)

[Compare Prices](#)

[Hardware Central](#)

[Talk to Us](#)

[Submit a URL](#)
[Suggest a Term](#)
[Report an Error](#)

International Domain Specialist

domain

internet.com

IT
Developer
Internet News
Small Business
Personal Technology
International

Search internet.com
Advertise
Corporate Info
Newsletters
Tech Jobs
E-mail Offers

Be a Commerce Partner
Business Gifts Canada
Online Booking Hotels
Logo Design
Mp3 Player Reviews
Health Insurance
Giveaways
Cars
Online Education
Graphics Cards
Laptop Battery
Merchant Accounts
Marketing Products
Business Lists
Memory

vehicle and in PDAs.

Based on the industry standard Unified Modeling Language (UML), Rational Rose Technical Developer provides a highly automated and reliable solution to the unique problems of concurrency and distribution.

Download: Compound Analysis for Java

A Linux-based Compound XML Document Editor using the an Eclipse Modeling Framework to provide a more customized editing experience for developers.

Download: Be Agile with DB2 on Rails

The Starter Toolkit for DB2 on Rails is a conveniently-packaged set of products and technologies that enables the quick creation of a configuration for building DB2 Web apps using Ruby on Rails technology.

Download: IBM Rational Data Architect V7.0

IBM Rational Data Architect is an enterprise data modeling and integration design tool designed to help data architects design relational and federated databases, understand data assets and their relationships and streamline database projects.

Download: IBM Rational Software Modeler V7.0

IBM Rational Software Modeler is a Unified Modeling Language (UML) 2.0-based visual modeling and design tool for architects, systems analysts, and designers who are responsible for specifying and maintaining system or software models and designs.

•E-mail this definition to a colleague•

Sponsored listings

Application-Specific Integrated Circuits - Offers a range of electronic components, including ASIC products, memory chips, microprocessors, diodes, and transistors.

ASIC - Electronic component supplier of obsolete components and semiconductors. Includes ASIC.

Asics Corporation Company Research - Find information on Asics Corporation with operations and products, financials, officers, competitors and more at Hoover's.

For internet.com pages about **ASIC**
CLICK HERE. Also check out the
following links!

LINKS

✳ = Great Page!

Sponsored listings

Search for ASIC & Semiconductors - Broad-line Distributor Web site features real-time stock status and pricing, online ordering, RFQ, technical support, product datasheets and photos.

Related Categories

[Handhelds](#)

[Integrated Circuits \(ICs\)](#)

[Microprocessors](#)

Related Terms

[chip](#)

[lab](#)

[labeled](#)

[integrated circuit](#)

[silicon](#)

(Webopedia)

**Give Us Your
Feedback**

**Shopping
ASIC Products**
Compare Products, Prices and
Stores

Shop by Category:
Books
13 Model Matches

Magazine and Newspaper Subscriptions

1 Model Matches

Printer Accessories

9 Model Matches

Wireless Access Points

1 Model Matches



Luggage

1 Store Offers

**YOUR DATA IS DOUBLING
EVERY 18 MONTHS.**

**WHERE DO YOU
STORE IT ALL?**

Send your data with the HP StorageWorks
All-in-One Storage System

☐ Internet.com Sitemap

☐ EarthWeb Sitemap

☐ DevX Sitemap

☐ MediaBistro Sitemap

☐ Graphics Sitemap

☐ Solutions

JupiterOnlineMedia.

Internet.com

EARTHWEB®



mediabistro.com

gutenberg.org

Search:

Find

Jupitermedia Corporation has two divisions: Jupiterimages and JupiterOnlineMedia

Jupitermedia Corporate Info

Copyright 2007 Jupitermedia Corporation All Rights Reserved.

[Legal Notices](#), [Licensing](#), [Reprints](#), & [Permissions](#), [Privacy Policy](#)

[Web Hosting](#) | [Newsletters](#) | [Tech Jobs](#) | [Shopping](#) | [E-mail Offers](#)

EXHIBIT 5

**IN THE UNITED STATES DISTRICT COURT
FOR THE EASTERN DISTRICT OF TEXAS
MARSHALL DIVISION**

MICROUNITY SYSTEMS ENGINEERING, INC.,	§	
	§	
Plaintiff,	§	
	§	CIVIL ACTION NO. 2-04-CV-120 (TJW)
v.	§	
	§	
DELL, INC. and INTEL CORPORATION,	§	
Defendants.	§	

MEMORANDUM OPINION AND ORDER

After considering the submissions and the arguments of counsel, the court issues the following order concerning the claim construction issues:

I. Introduction

In this patent infringement suit, Plaintiff Microunity Systems Engineering, Inc. accuses Defendants Dell, Inc. and Intel Corporation of infringing eight United States patents. U.S. Patent Nos. 5,742,840 (“the ‘840 patent”), 5,794,060 (“the ‘060 patent”), 5,809,321 (“the ‘321 patent”), 5,794,061 (“the ‘061 patent”), 6,584,482 (“the ‘482 patent”), 6,643,765 (“the ‘765 patent”), and 6,725,356 (“the ‘356 patent”) disclose a method and apparatus for the processing of multi-media digital communications. Collectively, these seven patents are referred to as the “media processor patents.”¹ The plaintiff also asserts U.S. Patent No. 5,630,096 (“the ‘096 patent”) against the defendants. The ‘096 patent discloses “a controller for maximizing throughput of memory requests from an external device to a synchronous DRAM.”

The invention disclosed in the ‘840, ‘060, ‘061, and ‘321 patents generally “relates to the field of communications processing, and more particularly, to a method and apparatus for real-time

¹ The ‘840, ‘060, ‘061, and ‘321 patents share a common specification. The ‘765 and ‘356 patents also share a common specification.

processing of multi-media digital communications.” ‘840 patent, col. 1, ll. 15-17. The ‘765 and ‘356 patents relate to general purpose processor architectures, and in particular, wide operand architectures. ‘765 patent, col. 1, ll. 19-21. These media processor patents disclose microprocessor designs that are capable of processing different types of media data, including audio, video, and graphics data, at very high volume in real-time.

II. Law Governing Claim Construction

“A claim in a patent provides the metes and bounds of the right which the patent confers on the patentee to exclude others from making, using or selling the protected invention.” *Burke, Inc. v. Bruno Indep. Living Aids, Inc.*, 183 F.3d 1334, 1340 (Fed. Cir. 1999). Claim construction is an issue of law for the court to decide. *Markman v. Westview Instruments, Inc.*, 52 F.3d 967, 970-71 (Fed. Cir. 1995) (en banc), *aff’d*, 517 U.S. 370 (1996).

To ascertain the meaning of claims, the court looks to three primary sources: the claims, the specification, and the prosecution history. *Markman*, 52 F.3d at 979. Under the patent law, the specification must contain a written description of the invention that enables one of ordinary skill in the art to make and use the invention. A patent’s claims must be read in view of the specification, of which they are a part. *Id.* For claim construction purposes, the description may act as a sort of dictionary, which explains the invention and may define terms used in the claims. *Id.* “One purpose for examining the specification is to determine if the patentee has limited the scope of the claims.” *Watts v. XL Sys., Inc.*, 232 F.3d 877, 882 (Fed. Cir. 2000).

Nonetheless, it is the function of the claims, not the specification, to set forth the limits of the patentee’s claims. Otherwise, there would be no need for claims. *SRI Int’l v. Matsushita Elec. Corp.*, 775 F.2d 1107, 1121 (Fed. Cir. 1985) (en banc). The patentee is free to be his own

lexicographer, but any special definition given to a word must be clearly set forth in the specification. *Intellicall, Inc. v. Phonometrics*, 952 F.2d 1384, 1388 (Fed. Cir. 1992). And, although the specification may indicate that certain embodiments are preferred, particular embodiments appearing in the specification will not be read into the claims when the claim language is broader than the embodiments. *Electro Med. Sys., S.A. v. Cooper Life Sciences, Inc.*, 34 F.3d 1048, 1054 (Fed. Cir. 1994).

This court's claim construction decision must be informed by the Federal Circuit's recent decision in *Phillips v. AWH Corporation*, 2005 WL 1620331 (Fed. Cir. July 12, 2005)(en banc). In *Phillips*, the court set forth several guideposts that courts should follow when construing claims. In particular, the court reiterated that "the *claims* of a patent define the invention to which the patentee is entitled the right to exclude." 2005 WL 1620331, at *4 (emphasis added)(quoting *Innova/Pure Water, Inc. v. Safari Water Filtration Systems, Inc.*, 381 F.3d 1111, 1115 (Fed. Cir. 2004)). To that end, the words used in a claim are generally given their ordinary and customary meaning. *Id.* at *5. The ordinary and customary meaning of a claim term "is the meaning that the term would have to a person of ordinary skill in the art in question at the time of the invention, i.e. as of the effective filing date of the patent application." *Id.* This principle of patent law flows naturally from the recognition that inventors are usually persons who are skilled in the field of the invention. The patent is addressed to and intended to be read by others skilled in the particular art. *Id.*

The primacy of claim terms notwithstanding, *Phillips* made clear that "the person of ordinary skill in the art is deemed to read the claim term not only in the context of the particular claim in which the disputed term appears, but in the context of the entire patent, including the specification." *Id.* Although the claims themselves may provide guidance as to the meaning of particular terms,

those terms are part of “a fully integrated written instrument.” *Id.* at **6-7 (*quoting Markman*, 52 F.3d at 978). Thus, the *Phillips* court emphasized the specification as being the primary basis for construing the claims. *Id.* at **7-8. As the Supreme Court stated long ago, “in case of doubt or ambiguity it is proper in all cases to refer back to the descriptive portions of the specification to aid in solving the doubt or in ascertaining the true intent and meaning of the language employed in the claims.” *Bates v. Coe*, 98 U.S. 31, 38 (1878). In addressing the role of the specification, the *Phillips* court quoted with approval its earlier observations from *Renishaw PLC v. Marposs Societa' per Azioni*, 158 F.3d 1243, 1250 (Fed. Cir. 1998):

Ultimately, the interpretation to be given a term can only be determined and confirmed with a full understanding of what the inventors actually invented and intended to envelop with the claim. The construction that stays true to the claim language and most naturally aligns with the patent’s description of the invention will be, in the end, the correct construction.

Consequently, *Phillips* emphasized the important role the specification plays in the claim construction process.

The prosecution history also continues to play an important role in claim interpretation. The prosecution history helps to demonstrate how the inventor and the PTO understood the patent. *Phillips*, 2005 WL 1620331 at *9. Because the file history, however, “represents an ongoing negotiation between the PTO and the applicant,” it may lack the clarity of the specification and thus be less useful in claim construction proceedings. *Id.* Nevertheless, the prosecution history is intrinsic evidence. That evidence is relevant to the determination of how the inventor understood the invention and whether the inventor limited the invention during prosecution by narrowing the scope of the claims.

Phillips rejected any claim construction approach that sacrificed the intrinsic record in favor

of extrinsic evidence, such as dictionary definitions or expert testimony. The *en banc* court condemned the suggestion made by *Texas Digital Systems, Inc. v. Telegenix, Inc.*, 308 F.3d 1193 (Fed. Cir. 2002), that a court should discern the ordinary meaning of the claim terms (through dictionaries or otherwise) before resorting to the specification for certain limited purposes. *Id.* at **13-14. The approach suggested by *Texas Digital*—the assignment of a limited role to the specification—was rejected as inconsistent with decisions holding the specification to be the best guide to the meaning of a disputed term. *Id.* According to *Phillips*, reliance on dictionary definitions at the expense of the specification had the effect of “focus[ing] the inquiry on the abstract meaning of words rather than on the meaning of the claim terms within the context of the patent.” *Id.* at *14. *Phillips* emphasized that the patent system is based on the proposition that the claims cover only the invented subject matter. *Id.* What is described in the claims flows from the statutory requirement imposed on the patentee to describe and particularly claim what he or she has invented. *Id.* The definitions found in dictionaries, however, often flow from the editors’ objective of assembling all of the possible definitions for a word. *Id.*

Phillips does not preclude all uses of dictionaries in claim construction proceedings. Instead, the court assigned dictionaries a role subordinate to the intrinsic record. In doing so, the court emphasized that claim construction issues are not resolved by any magic formula. The court did not impose any particular sequence of steps for a court to follow when it considers disputed claim language. *Id.* at *16. Rather, *Phillips* held that a court must attach the appropriate weight to the intrinsic sources offered in support of a proposed claim construction, bearing in mind the general rule that the claims measure the scope of the patent grant. The court now turns to a description of the technology in the case, followed by an assessment of the terms and phrases in dispute.

III. Discussion

The parties refer to seven of the patents as the media processor patents. In general, these patents describe an invention designed to consolidate the functions of several separate processors into a single processor for processing multi-media digital communications. As described in the abstract, the inventions relate to a general purpose, programmable media processor for processing and transmitting a media data stream of audio, video, radio, graphics, encryption, authentication, and networking information in real-time. '840 patent, Abstract. In the patents, the inventors noted certain shortcomings of application specific integrated circuits ("ASICs") as a solution to the efficient processing of broadband media data. The solution to using a number of different ASICs was to develop a single unified media processor. By combining the functionality of the various ASICs into a single processor, referred to as a media processor, the patentees sought to avoid the various problems associated with the use of a multiplicity of application-specific circuits.

The media processor incorporates the functionality of the different ASICs into a processor that can process the different media data types having different data sizes. The execution and arithmetic units can process different data types and sizes, at least in part, because of an ability referred to as "dynamic partitioning." The incoming data, which may be of different types, can be divided in accordance with the type of data it is. As the input data type changes (e.g. from video to audio), the data size may change, requiring a different partitioning. Because the execution and arithmetic units can process media data of different types and sizes, there is no time where a substantial amount of silicon "real estate" is idle. Moreover, the media processor is very flexible because the execution and arithmetic units are not limited to operating on any one particular data type, but can accommodate media data of different types and sizes. Finally, to maximize throughput

of the media processor, the execution and arithmetic units can perform operations on a group of data for each instruction, rather than each operation requiring a separate instruction.

In addition to the media processor patents, the plaintiff has asserted the '096 patent against the defendants. The '096 patent describes a memory controller for maximizing throughput of memory requests. The controller provides an interface between the memory and other devices, such as a processor. The controller enables reading data from, and writing data to, the memory by, for example, the processor. The controller can reorder memory requests and/or otherwise sort memory requests to make maximum use of the available opportunities for data transfer.

A. Disputed Terms of the Media Processor Patents

1. A plurality of media data streams

The first group of disputed claim terms involves the media data streams processed by the processor. The plaintiff contends that “a plurality of media data streams” means “two or more different types of media data streams such as audio, video, radio, graphics, encryption, authentication, and/or network information.” The defendants urge that “a plurality of media data streams” means “two or more concurrent streams of audio, video, radio, graphics, encryption, authentication, and/or network media data information from two or more sources.” At issue is the defendants’ use of the word “concurrent” in their proposed construction. After considering the submissions of counsel and the intrinsic record, the court is persuaded that the plaintiff’s position is correct. Accordingly, the court construes “a plurality of media data streams” to mean “two or more different types of media data streams.”

2. Unified media data streams

The court construes “unified media data streams” to mean “combined media data streams of

different types.”

3. Unified execution of multiple media data streams

The plaintiff proposes that “unified execution of multiple media data streams” be construed to mean “processing two or more different types of media data streams by the same execution unit.” On the other hand, the defendants submit that the disputed phrase means “operating on two or more media data streams in parallel, all at the same time with the same processor, without external specialized processors.” The parties’ primary dispute is whether media data streams are processed “in parallel,” “all at the same time,” and “without external processors.” The defendants correctly note that the general purpose media processor claimed in the patents was an improvement over the combination of prior art specialized processors; however, the court does not read the claim language or the specification to exclude all use of external processors. What is required is that the media processor has the capability of processing two or more different types of media data streams by the same execution unit. The court is persuaded that the plaintiff’s proposed construction is correct. Accordingly, the court construes “unified execution of multiple media data streams” to mean “processing two or more different types of media data streams by the same execution unit.”

4. Capable of dynamic partitioning

All of the asserted claims of the media processor patents recite the phrase “dynamic partitioning” or a similar limitation. The plaintiff urges that “capable of dynamic partitioning” means “able to divide the data into separate and distinct operands on an instruction by instruction basis.” The defendants contend that the disputed phrase means “capable of dividing width-wise into a variable number of elements for simultaneous parallel processing of any combination or permutation of media data types in any size.”

The parties' briefs and their hearing presentation focused on whether the court could (or should) consider an Appendix filed with the PTO in connection with this term. The court has concluded that resort to the Appendix is unnecessary, and that the specification is sufficiently illuminating to permit construction of this term. The specification of the '840 patent shows a Table I. Table I is an illustration of an instruction set for the media processor. That table illustrates data can be partitioned into byte-sized portions. Table I does not show partitioning of data on any basis other than a byte, or 8-bit level, nor does it show an instruction set capable of operating on data divided into different partition widths in the data path for any given 32-bits of data. Although Table I does not necessarily exclude a processor with the capability to partition data to the extent required by the defendants' proposed construction, Table I is more consistent with the plaintiff's proposed definition, as it illustrates precisely the type of dynamic partitioning that the plaintiff's construction permits. After considering the submissions of counsel and the intrinsic record, the court construes "capable of dynamic partitioning" to mean "capable of dividing width-wise into a variable number of elements."

5. Capable of dynamic partitioning based on the elemental width of data received from the data path, the elemental width being equal to or narrower than the data path

For the same reasons advanced above, the court construes "capable of dynamic partitioning based on the elemental width of data received from the data path, the elemental width being equal to or narrower than the data path" to mean "capable of dividing width-wise into a variable number of elements no wider than the data path, based upon the size of the data elements received from the data path."

6. Partitioning first and second registers into a plurality of floating point operands, said floating point operands having a defined bit width, wherein said defined bit width is dynamically variable

The court construes “partitioning first and second registers into a plurality of floating point operands, said floating point operands having a defined bit width, wherein said defined bit width is dynamically variable” to mean “dividing a first and a second register width-wise into a variable number of floating point operands based upon a variable width of the floating point element.”

7. Dynamically partition data received from the data path to account for an elemental width of the data wherein the elemental width of the data is equal to or narrower than the data path

The court construes “dynamically partition data received from the data path to account for an elemental width of the data wherein the elemental width of the data is equal to or narrower than the data path” to mean “dividing width-wise into a variable number of elements no wider than the data path, based upon the size of the data elements received from the data path.”

8. Dynamically partitionable arithmetic unit

The court construes “dynamically partitionable arithmetic unit” to mean “the arithmetic unit can be divided into a variable number of elements.”

9. Unified media processing

The court now turns to the disputed terms relating to the media processor limitations. The plaintiff urges that “unified media processing,” as used in the preamble, is not a limitation. Rather, the plaintiff contends that this is a statement of intended use. The defendants contend, however, that “unified media processing” is a limitation, and means “processing a media data stream using parallel processing and utilizing the entire width of the data path through dynamic partitioning, without

external specialized processors.”

After considering the submissions of counsel, the court concludes that “unified media processing” is not a claim limitation. The phrase “unified media processing” appears in the preamble of claim 1 of the ‘321 patent, which provides “[a] system for unified media processing, comprising . . .” ‘321 patent, claim 1. The rule in the Federal Circuit is that “[i]n general, a preamble limits the invention if it recites essential structure or steps, or if it is ‘necessary to give life, meaning, and vitality’ to the claim.” *Catalina Mktg. Int’l v. Coolsavings.com*, 289 F.3d 801, 808 (Fed. Cir. 2002). The phrase “unified media processing” does not appear in the body of the claim, is not necessary to give life, meaning and vitality to the claim, and does not provide an antecedent basis for the term “media processor,” which does appear in the claim. The court therefore concludes that “unified media processing” is not a claim limitation. No construction of this phrase is necessary.

10. General purpose media processor/ General purpose programmable media processor/ Media processor/ Programmable media processor/ General purpose [multiple precision parallel operation] programmable media processor

Next, the court turns to the construction of the media processor terms. The plaintiff contends that the “general purpose media processor” is “a general purpose processor that also does media processing.” The defendants urge that “general purpose media processor” means “a single programmable processor that processes multiple media data streams without external specialized processors.” There are two disputes between the parties regarding the construction of this term and the remaining “media processor” terms. The primary dispute between the parties is whether the “media processor” operates without “external specialized processors.” A second issue is whether the term “programmable media processor” is a claim limitation.

After considering the submissions and arguments of counsel, the court construes each of the disputed phrases above to mean “a processor having an execution unit capable of operating on different media types and data sizes.” The court is not persuaded that the claim language or the specification necessarily excludes the presence of other, external processors, from the scope of the claim, as long as the processor itself has the requisite capabilities. With respect to the term “programmable media processor,” the court concludes that this term is not a claim limitation for the same reasons provided with respect to the term “unified media processing.”

11. Multi-precision arithmetic unit

The parties largely agree on the construction of “multi-precision arithmetic unit” with one exception. The plaintiffs contend that the “multi-precision arithmetic unit” is “a unit that can perform addition, subtraction, multiplication, division, and other integer and floating point arithmetic operations on data streams of varying sizes.” The defendants object to the plaintiff’s proposed construction on two grounds. First, the defendants contend that the term “unit” refers to a defined circuit block, and not circuitry distributed across the media processor, as the plaintiff argues. Second, the defendants further contend that the language “other integer and floating point arithmetic operations” should not be included in the construction. Thus, the defendants propose the following construction of “multi-precision arithmetic unit” – “a unit that can perform addition, subtraction, multiplication, division, and other arithmetic operations on data streams of varying sizes.” The ‘840 patent provides as follows:

Many of the logic blocks themselves can also be replaced [sic] with a single multi-precision arithmetic unit, which can be internally partitioned under software control to perform addition, multiplication, division, and other integer and floating point arithmetic operations on symbol streams of varying widths while sustaining the full data throughput of the memory hierarchy.

'840 patent, col. 2, ll. 58-65. Based on the cited portion of the specification, the court is persuaded that the plaintiff's construction is correct and adopts it. The court declines to further define "unit" to require a single circuit block.

12. Multi-precision execution unit

The court construes "multi-precision execution unit" to mean "a unit that receives instructions and executes the instructions to perform simultaneous parallel operations on the plurality of media data streams, each of a width up to the width of the data path."

13. Operable to perform unique operations on each component symbol

The plaintiff proposes that "operable to perform unique operations on each component symbol" should be construed to mean "capable of performing a distinct operation on each component of a data unit." The defendants urge that the disputed phrase means "able to simultaneously perform different operations on each partitioned item of data." At issue is the meaning of the term "unique." The plaintiff contends it is sufficient that the multi-precision execution unit performs the same, single chosen type of operation (*e.g.*, multiply) on each component symbol, albeit in different, separate, and distinct instances." *See* Plaintiff's Reply Brief at 33. The defendants insist, however, that the plaintiff's construction should be rejected because it does not reflect the definition of the term "unique." After considering the submissions of counsel, the court concludes that the plaintiff is correct. The court defines this term to mean "operable to perform unique operations on each component symbol" to mean "capable of performing a distinct operation on each component of a data unit."

14. A switch coupled to the data path and programmable to manipulate data received from the data path, the switch providing data streams to the data path

The plaintiff asserts that this phrase should be construed to mean the following: “a routing device that is: (1) coupled to and receives data from the data path, (2) rearranges the data fields received from the data path in different ways in response to instructions by performing operations such as deals, shuffles, shifts, expands, compresses, swizzles, permutes, and reverses, and (3) provides the rearranged data fields to the data path.” The defendants, however, urge that the disputed phrase should be construed more broadly to “hardware and/or software that performs data handling operations on unified media streams.” They argue that the plaintiff’s proposed construction imports limitations from the specification by requiring that the switch perform operations such as “deals, shuffles, shifts, expands, compresses, swizzles, permutes, and reverses.” In the context of these patents, the court construes “a switch coupled to the data path and programmable to manipulate data received from the data path, the switch providing data streams to the data path” to mean “a routing device that is: (1) coupled to and receives data from the data path, (2) rearranges the data fields received from the data path in different ways in response to instructions, and (3) provides the rearranged data fields to the data path.”

15. Manipulating component fields

The court construes “manipulating component fields” to mean “rearranging the data fields received from the data path in different ways.”

16. Group data handling operations

The court construes “group data handling operations” to mean “data handling operations applied to a group of partitioned fields.”

17. Register controllable cross bar switch

The plaintiff asserts that “register controllable cross bar switch” means “a routing device that

selectively couples a plurality of outputs to a plurality of inputs under the control of the contents of a register.” The defendants propose that “register controllable cross bar switch” means “a switch which can independently connect any input to any output, and that is controlled through the use of hardware storage locations in the media processor that are available to the user/programmer.” At issue is whether the “cross bar switch” must be able to connect any input with any output.

The plaintiff contends that “switch 104,” which is described in the specification of the ‘061 patent, is the cross-bar switch recited in the claims. According to the plaintiff, the specification does not require that “switch 104” be able to connect any input with any output. The defendants, on the other hand, argue that “cross bar switch” is a term of art that means “a switch that allows any input to be connected to any output.” The defendants note that the term “cross bar” does not appear in the specifications of the media processor patents and that there is no indication that the patents use the term “cross bar” in a manner different from its ordinary meaning. After considering the submissions of counsel, the court construes “register controllable cross bar switch” to mean “a routing device that selectively couples a plurality of outputs to a plurality of inputs under the control of the contents of a register.”

18. Extended mathematical element

The plaintiff proposes that “extended mathematical element” be construed to mean “a unit that performs additional mathematical operations that are specialized operations for efficient media processing.” The defendants object to the plaintiff’s use of the language “specialized operations for efficient media processing.” They also contend that the specification of the ‘840 patent provides that “operations performed by the extended mathematical unit are ‘higher level’ than those performed by the ALU . . .” Defendants’ Sur-reply Brief at 13. Thus, the defendants urge that

“extended mathematical element” be construed to mean “a unit that performs higher level mathematical operations than the arithmetic unit.” After considering the submissions of counsel, the court construes “extended mathematical element” to mean “a unit that performs additional mathematical operations other than addition, subtraction, multiplication, division, and other floating point operations.”

19. An extended mathematical element coupled to the data path and programmable to implement additional mathematical operations at substantially peak data throughput

The court construes “an extended mathematical element coupled to the data path and programmable to implement additional mathematical operations at substantially peak data throughput” to mean “a programmable unit coupled to the data path that performs additional mathematical operations other than addition, subtraction, multiplication, division, and other floating point operations at substantially peak data throughput.”

20. Table look-up . . . [operation]

The court construes “table look-up . . . [operation]” to mean “an operation that uses a known value to locate an unknown value in a table.”

21. Storing the unified media data streams in a general register file

The plaintiff submits that “storing the unified media data streams in a general register file” means “storing the unified media data streams in a set of registers, which may be addressed by their number in the set, in which the registers can be used for various purposes.” The defendants assert that the disputed phrase means “storing the unified media data streams in a set of hardware storage locations that are available to the user/programmer for a wide variety of functions.” Primarily at issue is whether “a general register file” must be available for a “wide variety” of functions.

The plaintiff contends that the “general register file” need not be available for a wide variety of functions, but must be available for “different” or “various” purposes. The plaintiff asserts that its proposed construction is supported by the specifications of the ‘060 and ‘840 patents and that its construction would be understood by one ordinarily skilled in the art. According to the plaintiff, the patent specifications only require that the registers not be dedicated or specific purpose registers. The defendants, on the other hand, contend that the media processor patents use the term “general register file” consistent with its ordinary meaning, which requires that the register file be available for “a wide variety of functions.” After considering the submissions of counsel, the court construes “storing the unified media data streams in a general register file” to mean “storing the unified media data streams in a set of hardware storage locations that are available to the user/programmer for various purposes.”

22. Multiple operands in partitioned fields of operand registers

The court construes “multiple operands in partitioned fields of operand registers” to mean “more than one object upon which operations are performed, each object being stored in a separate and distinct field of a register.”

23. Storing partitioned data in registers

The plaintiff contends that “storing partitioned data in registers” means “storing in registers the data that was divided into separate and distinct data fields.” The defendants, on the other hand, argue that “storing partitioned data in registers” means “the results of the dynamic partitioning of the data stream is [sic] stored in adjacent portions of registers.” The plaintiff disagrees with the defendant’s proposed construction because it requires that the results of the dynamic partitioning be stored in adjacent portions of the register. According to the plaintiff, the specifications of the media

processor patents disclose storing partitioned data in non-adjacent portions. The defendants contend that a person of ordinary skill in the art would understand that the data are stored in adjacent portions of registers. After considering the arguments of counsel, the court construes “storing partitioned data in registers” to mean “the results of the dynamic partitioning of the data stream are stored in registers.”

24. High bandwidth external interface

The plaintiff argues that “high bandwidth external interface” means “an interface between the media processor and external sources of data capable of operating at or near a rate that maintains substantially peak operation of the media processor.” The defendants urge that “high bandwidth external interface” means “an interface between the media processor and external sources of data that operates at or near the peak data throughput rate of the execution units [sic] of the media processor.” The court construes “high bandwidth external interface” to mean “an interface between the media processor and external sources of data that is capable of operating at or near the peak data throughput rate of the execution unit of the media processor.”

25. A high bandwidth external interface operable to receive a plurality of data of various sizes from an external source and communicate the received data over the data path at a rate that maintains substantially peak operation of the media processor

After considering the submissions of counsel, the court concludes that this phrase requires no additional construction.

26. High bandwidth interface

After considering the submissions of counsel, the court concludes that this phrase requires no additional construction.

27. Substantially peak rates

The court construes “substantially peak rates” to mean “simultaneous parallel processing using all or nearly all of the entire width of the data path.”

28. Data path

The term “data path” is construed to mean “the buses and circuit elements that convey data.”

29. Bi-directional communication fabric

The phrase “bi-directional communication fabric” means “an interprocessor communications network allowing communication in both directions.” The specification suggests that the patentee used “network” and “fabric” interchangeably. *See* ‘840 patent, col. 6, ll. 16-19. At least one programmable media processor is provided within the communications network for receiving, processing, and transmitting the at least one stream of unified media data over the bi-directional communications fabric.

30. Being capable of being represented by a defined bit width which is equal to said defined bit width of said operands

The plaintiff proposes that this phrase means “able to be represented by a data field of the same size as the floating point operands.” The defendants contend that the disputed phrase means “the bit width of the product of the permissible floating point operands must be no greater than the bit width of each of the operands.” The plaintiff objects to the defendants’ construction because of the limitation “no greater than the bit width of each of the operands.” The claim language, the plaintiff argues, requires the capability of being represented by a bit width *equal to* the bit width of the operands. The defendants insist, however, that their “no greater than” limitation is entirely consistent with the claim language and is found in the specification of the ‘482 patent. After

considering the submissions of counsel, the court adopts the plaintiff's construction and construes "being capable of being represented by a defined bit width which is equal to said defined bit width of said operands" accordingly.

31. Group floating point operations

The court construes "group floating point operations" to mean "floating point operations applied to a group of partitioned operands."

32. Dedicated memory

The term "dedicated memory" appears in all of the claims of the '321 patent. In its reply brief, the plaintiff proposes that "dedicated memory" should be construed to mean "memory that is within the media processor that is accessible only through memory circuitry associated with the media processor." The defendants contend that under the plaintiff's construction, the memory is not dedicated because it is accessible by circuitry associated with the media processor and associated with another processor. Thus, the defendants insist that "dedicated memory" means "memory that is within the media processor that is accessible only through the media processor." Mindful that the claim language requires a "dedicated" memory, the court adopts the defendants' construction of "dedicated memory" and construes this term accordingly.

33. Boolean . . . mathematical operation

The court construes "Boolean . . . mathematical operation" as follows: "A Boolean operation is an operation that applies formal logic (for example, 'AND,' 'OR,' 'NOR,' etc.)."

B. Disputed Terms of the '096 Patent

1. Throughput maximizing unit for processing said memory requests to the synchronous DRAM in response to scheduling which maximizes the use of data slots by the synchronous DRAM

The court first considers whether § 112 ¶ 6 applies. The plaintiff asserts that it does not and that the phrase “throughput maximizing unit for processing said memory requests . . .” means “an element of the controller that processes memory requests in response to scheduling constraints of the synchronous DRAM which maximizes the use of data slots.” In their response brief, the defendants contend that “maximizing throughput of said memory requests . . .” is a means-plus-function limitation. The defendants argue that “claims 1 and 3 fail to recite any structure for processing memory requests to maximize the use of data slots, and claims 11, 13, and 20 fail to recite any acts prescribing how to maximize the use of data slots.” Defendants’ Response Brief at 71. The absence of the word “means” raises a presumption that § 112, ¶ 6 does not apply. The court concludes that “throughput maximizing unit for processing said memory requests . . .” is not a means-plus-function limitation. The court therefore adopts the plaintiff’s construction and construes “throughput maximizing unit for processing said memory requests” as an “element of the controller that processes memory requests in response to scheduling constraints of the synchronous DRAM which maximizes the use of data slots.”

2. Throughput maximizing unit

The plaintiff proposes that the term “throughput maximizing unit” is “an element of the controller that processes memory requests in response to scheduling constraints of the synchronous DRAM.” The defendants contend that the term “throughput maximizing unit” is a means-plus-function limitation. After considering the arguments of counsel, the court concludes that the term

“throughput maximizing unit” is not a means-plus-function limitation. Accordingly, the court construes “throughput maximizing unit” to mean “an element of the controller that processes memory requests in response to scheduling constraints of the synchronous DRAM, which maximizes throughput.”

3. Maximizing throughput of said memory requests to the synchronous DRAM so that use of the data slots by the synchronous DRAM is maximized

The court adopts the plaintiff’s construction of “maximizing throughput of said memory requests . . .” and construes this phrase to mean “scheduling memory requests to the synchronous DRAM to maximize throughput so that the use of data slots is maximized.”

4. Memory requests

The term “memory requests” appears in all of the asserted claims of the ‘096 patent. The plaintiff proposes that “memory requests” are “requests from an external device, such as a processor, to a memory device.” The defendants assert that “memory requests” are “requests from an external device such as a processor, to load data from or store data to the synchronous DRAM.” The parties dispute whether “memory requests” must be directed to the synchronous DRAM. According to the defendants, “memory requests” are addressed only to the synchronous DRAM and not any other memory device. After considering the submissions of counsel, the court adopts the plaintiff’s construction of “memory requests.”

5. Data slots

The term “data slots” is recited in all of the asserted claims of the ‘096 patent. The plaintiff submits that “data slots” are “times during which data may be transferred to or from the SDRAM.” The defendants propose that “data slots” are “SDRAM clock cycles for transferring data.” The

plaintiff contends that the defendants propose “a highly specific construction of data slot related to the SDRAM clock cycle” that is not supported by or disclosed in the specification. However, the defendants argue that the ‘096 patent makes clear that the “data slots correspond to SDRAM clock cycles in Figs. 4(a -c) and does not disclose any other time period from which a data slot may be defined.” Defendants’ Sur-reply Brief at 20. After considering the submissions of counsel, the court construes the term “data slots” to mean “SDRAM clock cycles available for transferring data.”

6. Interfacing a processing device with a synchronous DRAM

The plaintiff proposes that “interfacing a processing device with a synchronous DRAM” means “reading and writing to the synchronous DRAM by a processing device.” The defendants urge that the disputed phrase means “translating memory requests into synchronous DRAM commands.” At issue is whether “interfacing” requires translation. In its reply brief, the plaintiff argues that the defendants’ proposed construction introduces a notion of translation, which is inconsistent with the ordinary meaning of the term “interfacing.” On the other hand, the defendants contend that the disclosed controller receives memory requests and translates them into SDRAM commands. The court construes the phrase “interfacing a processing device with a synchronous DRAM” to mean “enabling reading and writing to the synchronous DRAM by a processing device.”

7. Sorting said memory requests based on their addresses

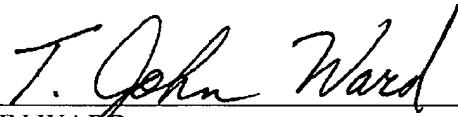
The plaintiff urges that “sorting said memory requests based on their addresses” means “segregating memory requests into one or more groups based on their addresses.” The defendants argue that the disputed phrase means “segregating memory requests into two or more groups according to their addresses.” At issue is the number of groups into which “memory requests” can be segregated. The court adopts the plaintiff’s construction of “sorting said memory requests based

on their addresses” and construes this phrase accordingly.

8. Means for developing memory requests from the processing device

This means-plus-function limitation appears in claim 10 of the ‘096 patent. The parties agree that the court need only construe the corresponding structure. The plaintiff contends that the corresponding structure is “bank sort unit 10 and equivalent structures.” The defendants urge that the corresponding structure is “command update unit 210, bank qualification unit 220, and Tables 1-2.” After considering the submissions of counsel, the court concludes that the corresponding structure consists of “bank sort unit 10 and equivalent structures.”

SIGNED this 26th day of August, 2005.



T. JOHN WARD
UNITED STATES DISTRICT JUDGE

EXHIBIT 6



US005794060A

United States Patent [19]

Hansen et al.

[11] Patent Number: 5,794,060
 [45] Date of Patent: Aug. 11, 1998

[54] **GENERAL PURPOSE, MULTIPLE PRECISION PARALLEL OPERATION, PROGRAMMABLE MEDIA PROCESSOR**

[75] Inventors: Craig Hansen, Los Altos; John Moussouris, Sunnyvale, both of Calif.

[73] Assignee: Microunity Systems Engineering, Inc., Sunnyvale, Calif.

[21] Appl. No.: 754,826

[22] Filed: Nov. 22, 1996

Related U.S. Application Data

[62] Division of Ser. No. 516,036, Aug. 16, 1995, Pat. No. 5,742,840.

[51] Int. Cl.⁶ G06F 9/00

[52] U.S. Cl. 395/800.01

[58] Field of Search 395/800.01, 670, 395/376, 280; 364/131-134, 736, 741, 745, 748, 754, 761, 768, 736.01, 745.01, 728.01, 754.01

[56] References Cited

U.S. PATENT DOCUMENTS

4,893,267	1/1990	Alsup et al.	364/745
4,975,868	12/1990	Freerksen	364/748
5,201,056	4/1993	Daniel et al.	395/800
5,268,855	12/1993	Mason et al.	364/748
5,426,600	6/1995	Nakagawa et al.	364/764

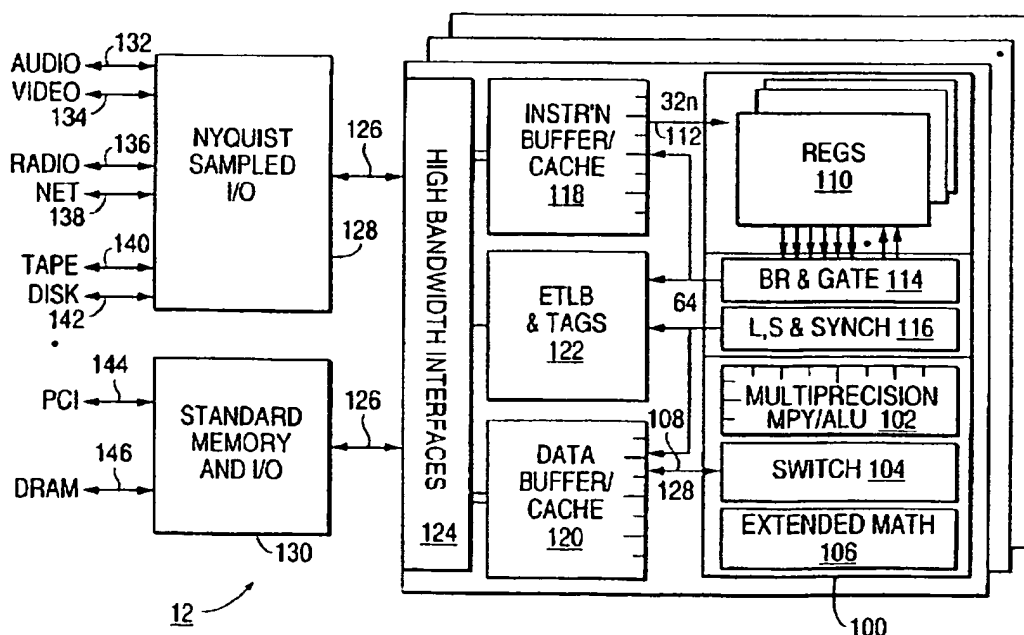
Primary Examiner—Alpesh M. Shah
 Attorney, Agent, or Firm—McDermott, Will & Emery

[57] ABSTRACT

A general purpose, programmable media processor for processing and transmitting a media data stream of audio, video, radio, graphics, encryption, authentication, and networking information in real-time. The media processor incorporates an execution unit that maintains substantially peak data throughout of media data streams. The execution unit includes a dynamically partitionable multi-precision arithmetic unit, programmable switch and programmable extended mathematical element. A high bandwidth external interface supplies media data streams at substantially peak rates to a general purpose register file and the multi-precision execution unit. A memory management unit, and instruction and data cache/buffers are also provided. High bandwidth memory controllers are linked in series to provide a memory channel to the general purpose, programmable media processor. The general purpose, programmable media processor is disposed in a network fabric consisting of fiber optic cable, coaxial cable and twisted pair wires to transmit, process and receive single or unified media data streams. Parallel general purpose media processors are disposed throughout the network in a distributed virtual manner to allow for multi-processor operations and sharing of resources through the network. A method for receiving, processing and transmitting media data streams over the communications fabric is also provided.

4 Claims, 25 Drawing Sheets

Microfiche Appendix Included
 (4 Microfiche, 387 Pages)



U.S. Patent

Aug. 11, 1998

Sheet 1 of 25

5,794,060

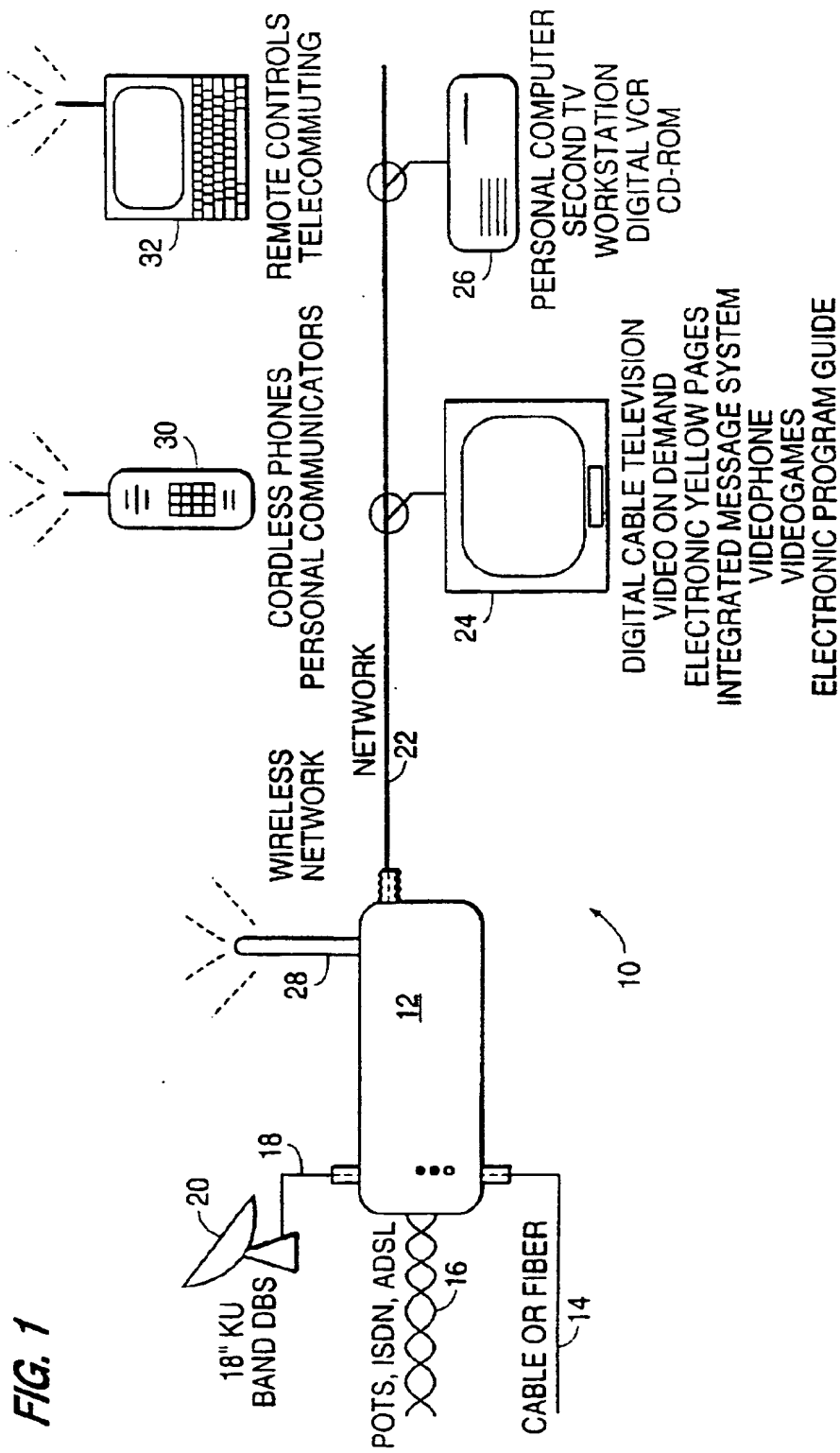
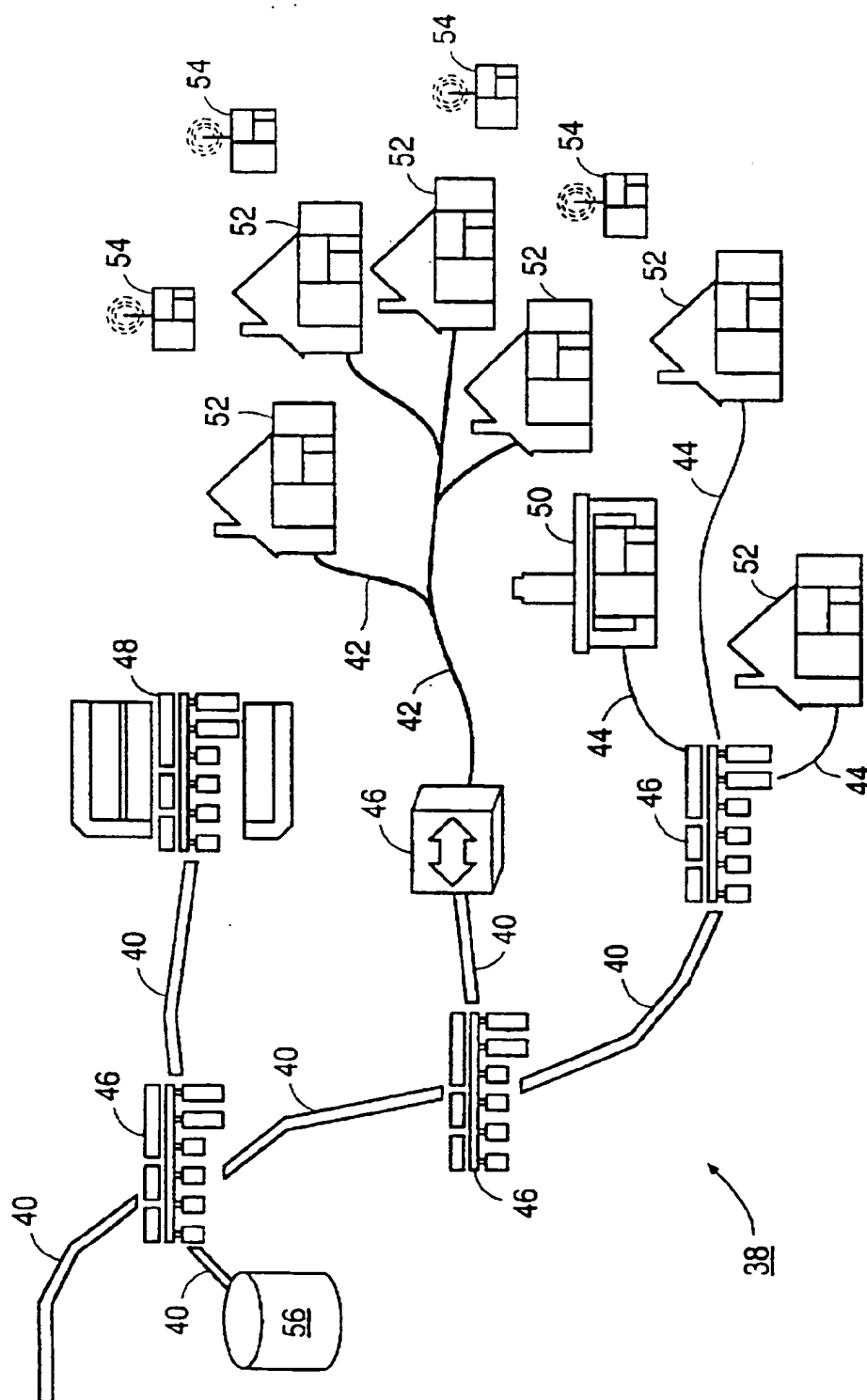


FIG. 2

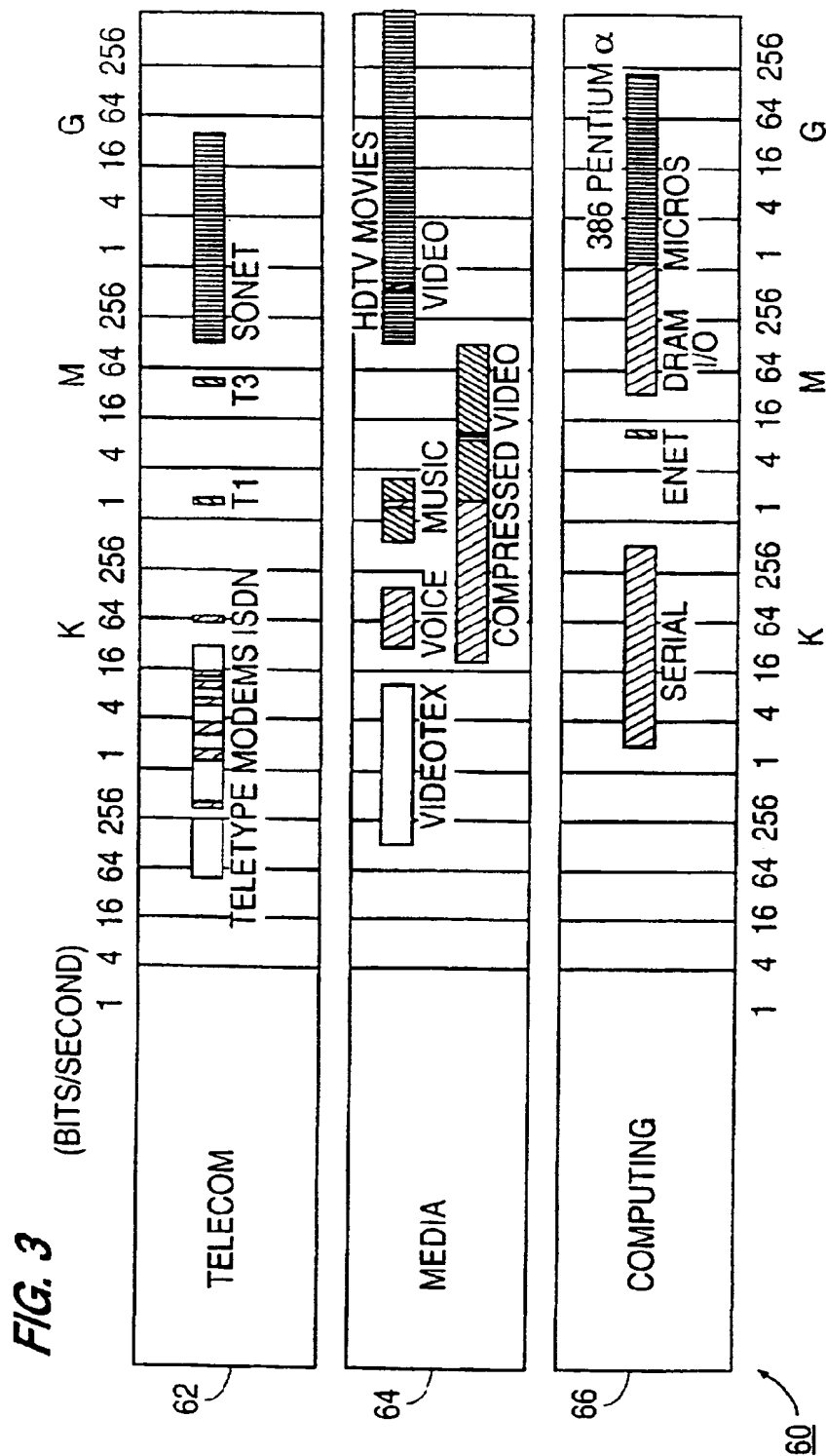


U.S. Patent

Aug. 11, 1998

Sheet 3 of 25

5,794,060

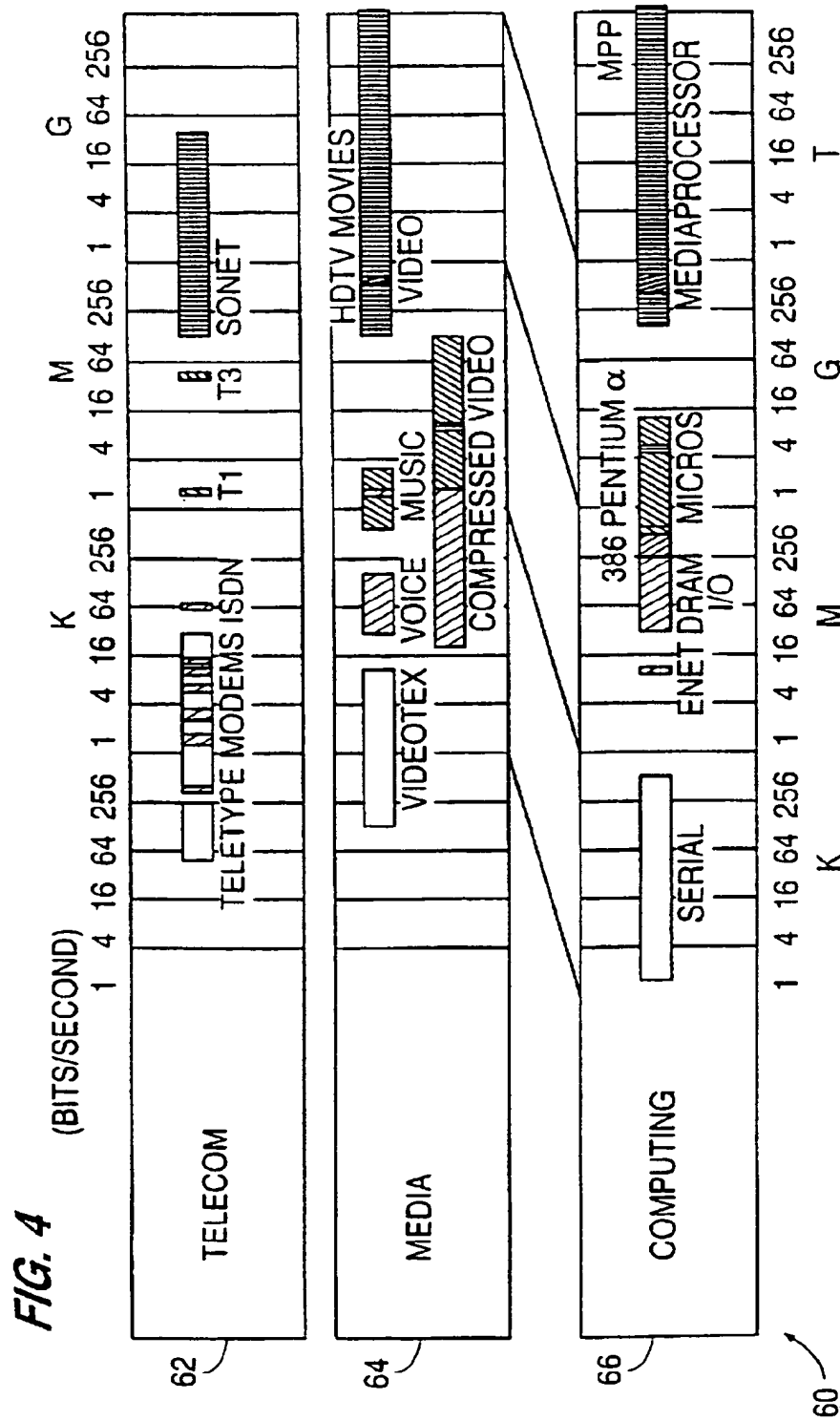


U.S. Patent

Aug. 11, 1998

Sheet 4 of 25

5,794,060



U.S. Patent

Aug. 11, 1998

Sheet 5 of 25

5,794,060

FIG. 5

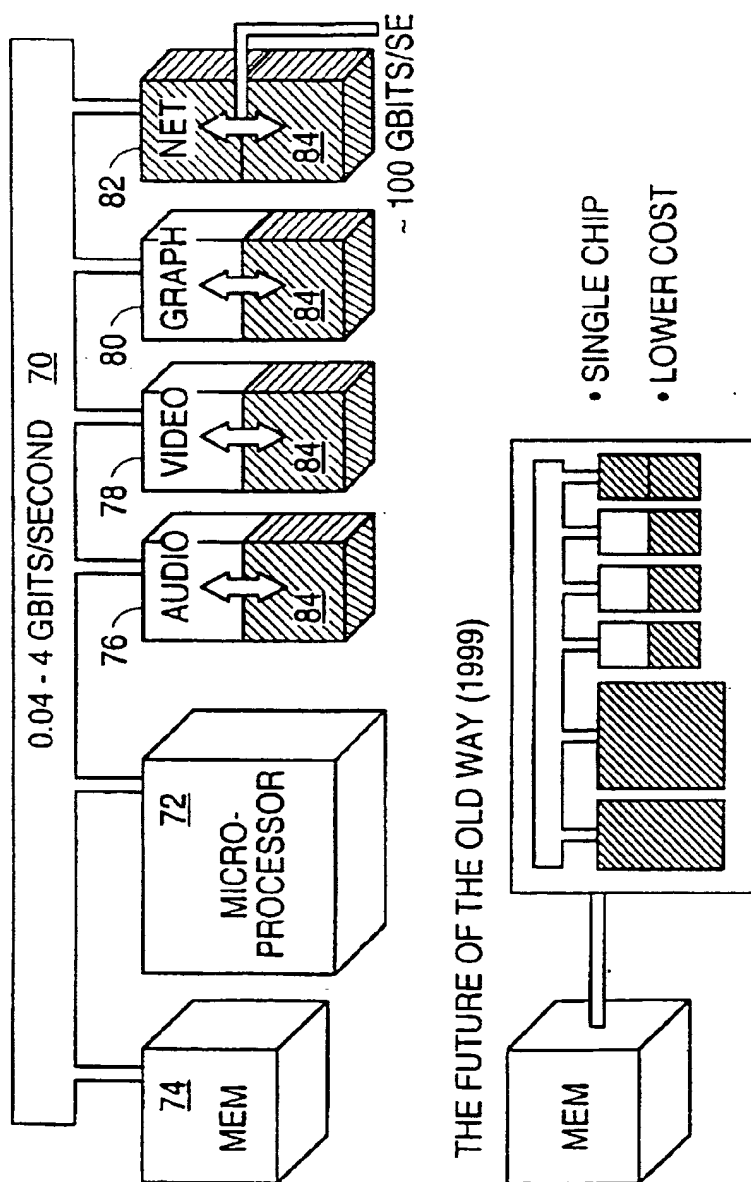
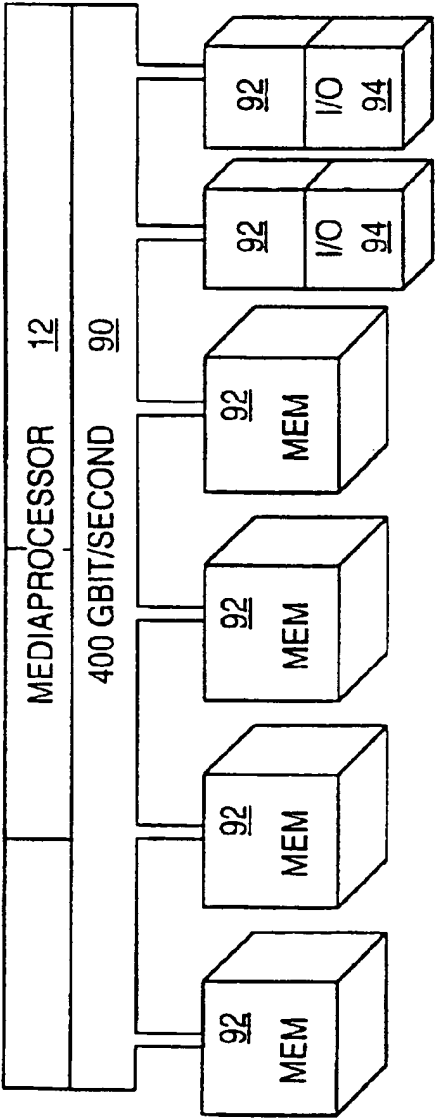
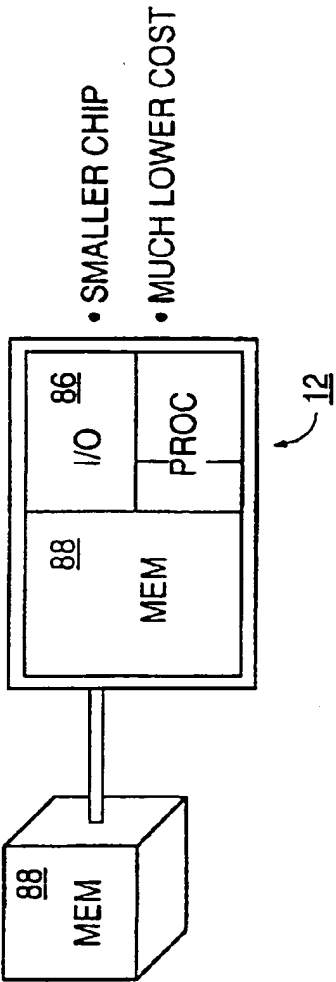


FIG. 6



THE FUTURE OF THE UNIFIED WAY (1995)



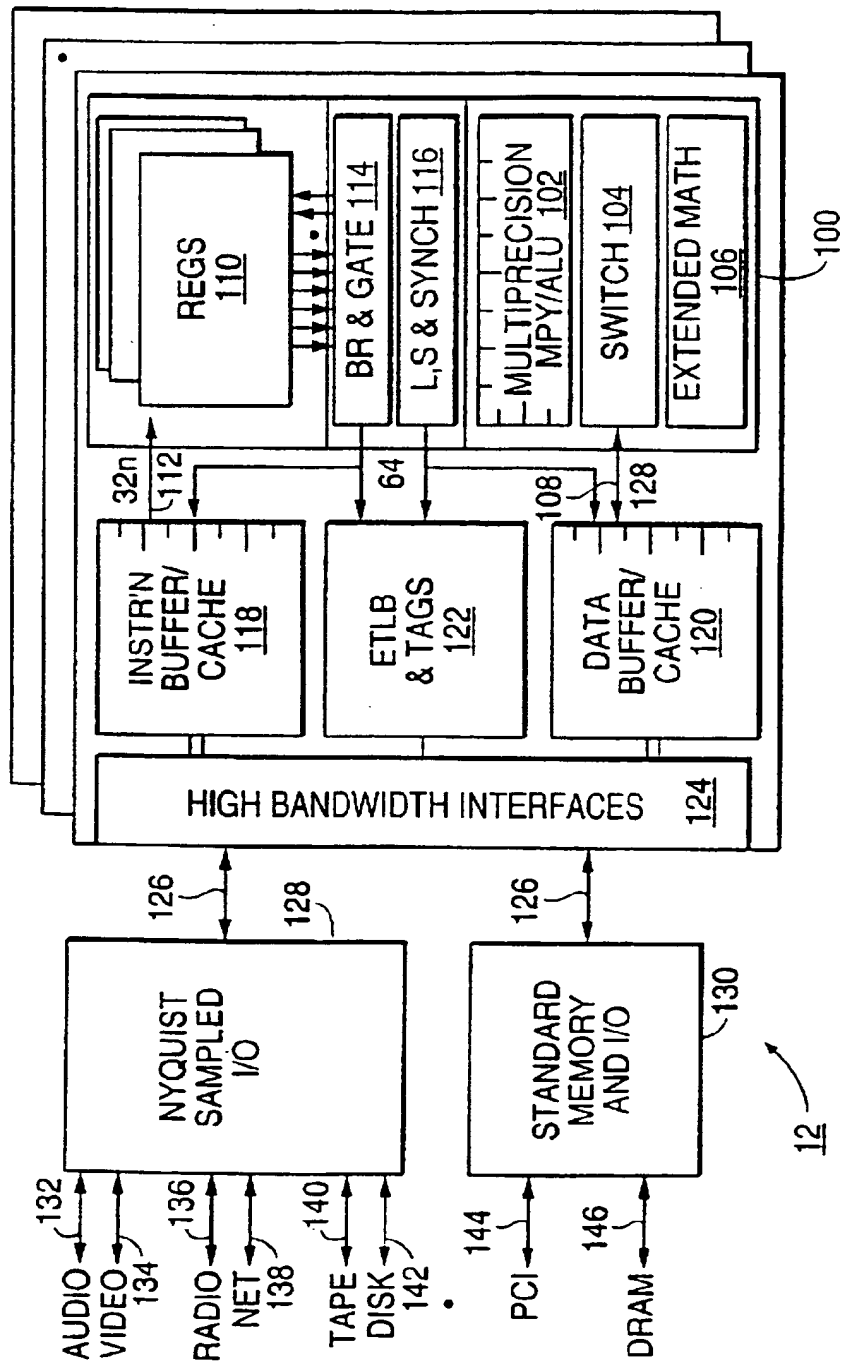
U.S. Patent

Aug. 11, 1998

Sheet 7 of 25

5,794,060

FIG. 7



U.S. Patent

Aug. 11, 1998

Sheet 8 of 25

5,794,060

FIG. 8(a)

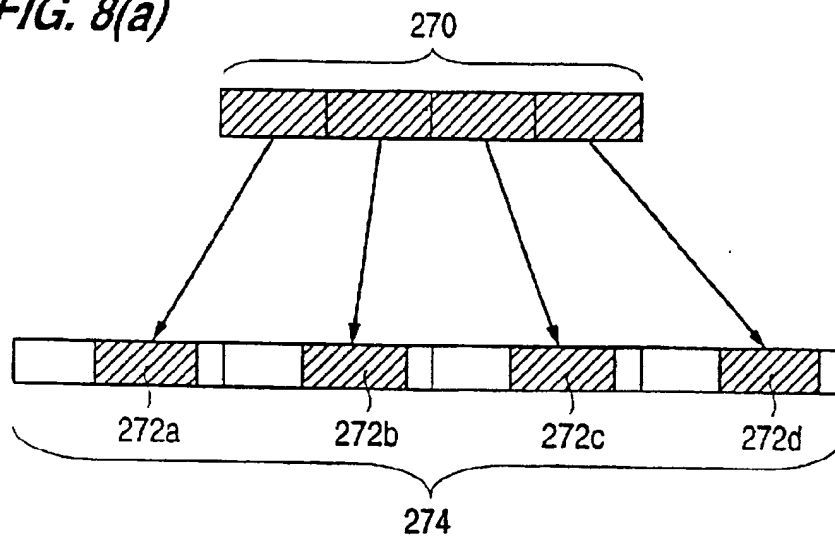
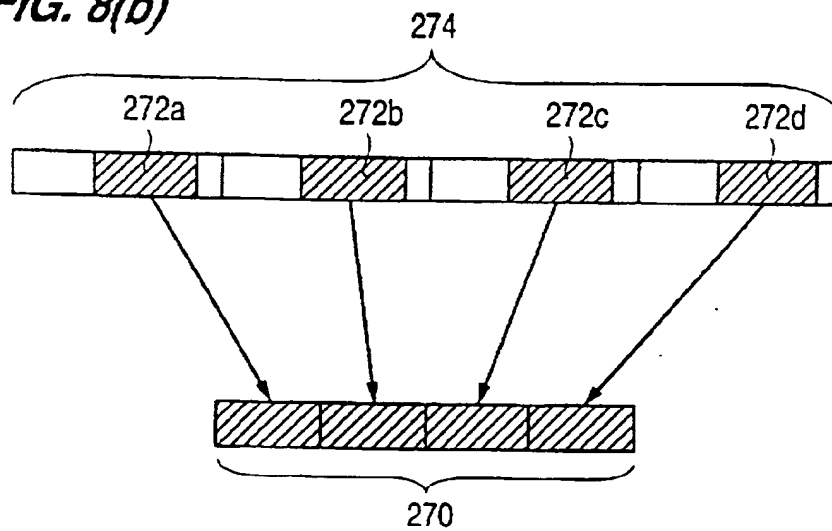


FIG. 8(b)



U.S. Patent

Aug. 11, 1998

Sheet 9 of 25

5,794,060

FIG. 8(c)

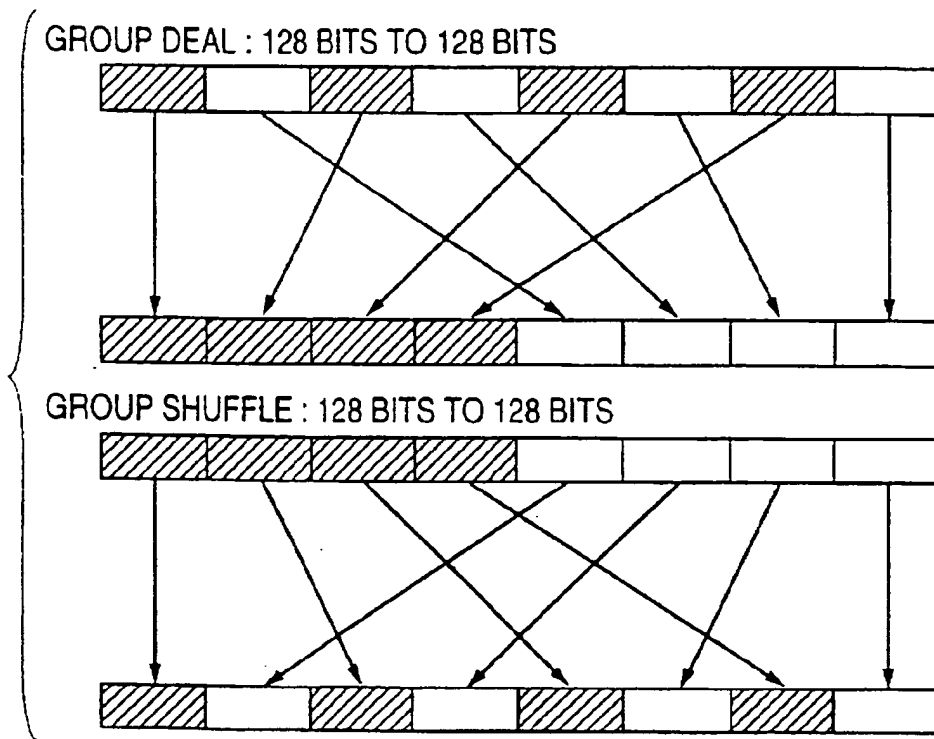
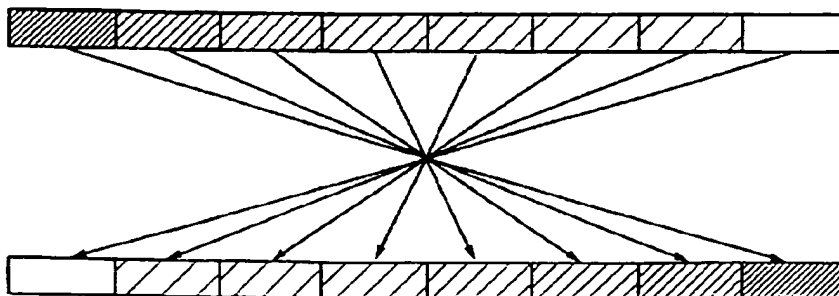


FIG. 8(d)



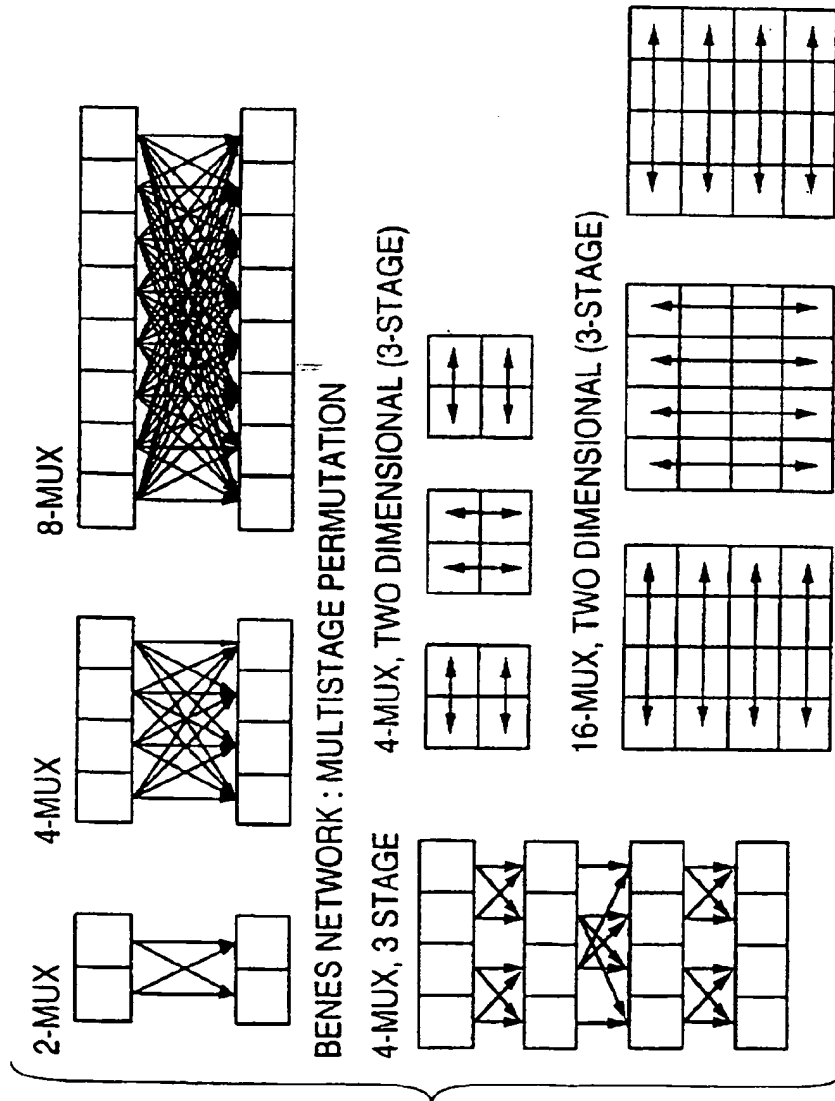
U.S. Patent

Aug. 11, 1998

Sheet 10 of 25

5,794,060

FIG. 8(e)



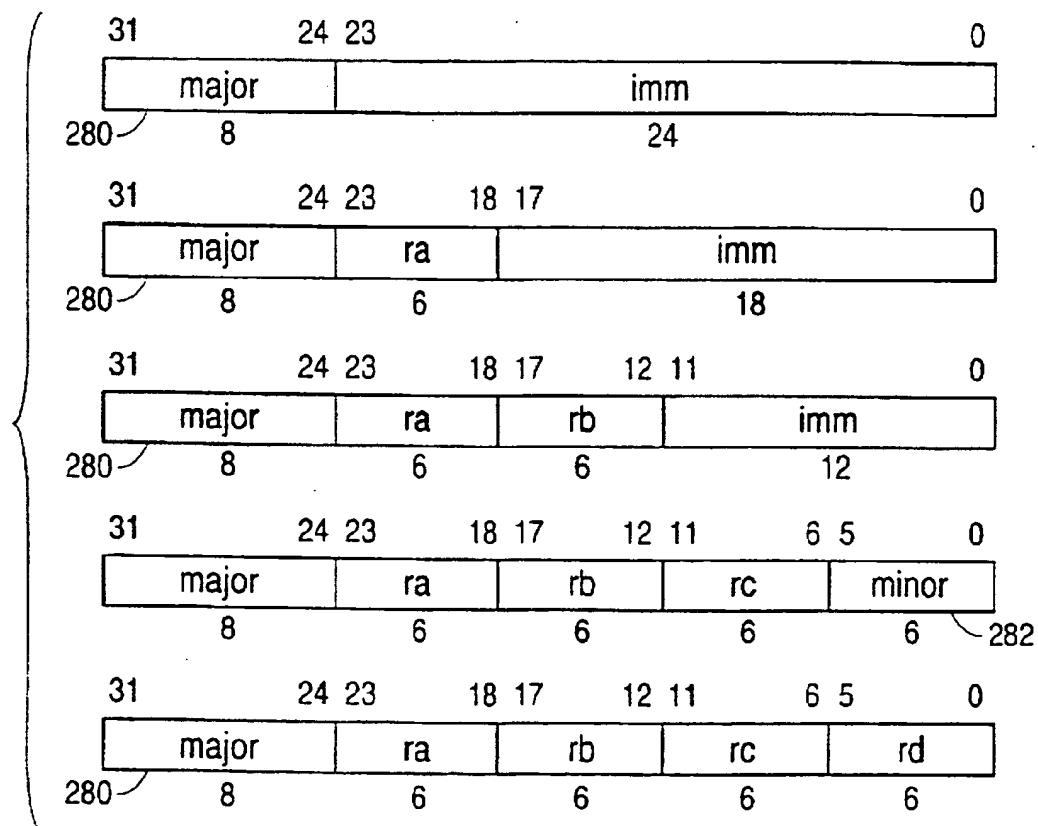
U.S. Patent

Aug. 11, 1998

Sheet 11 of 25

5,794,060

FIG. 9(a)



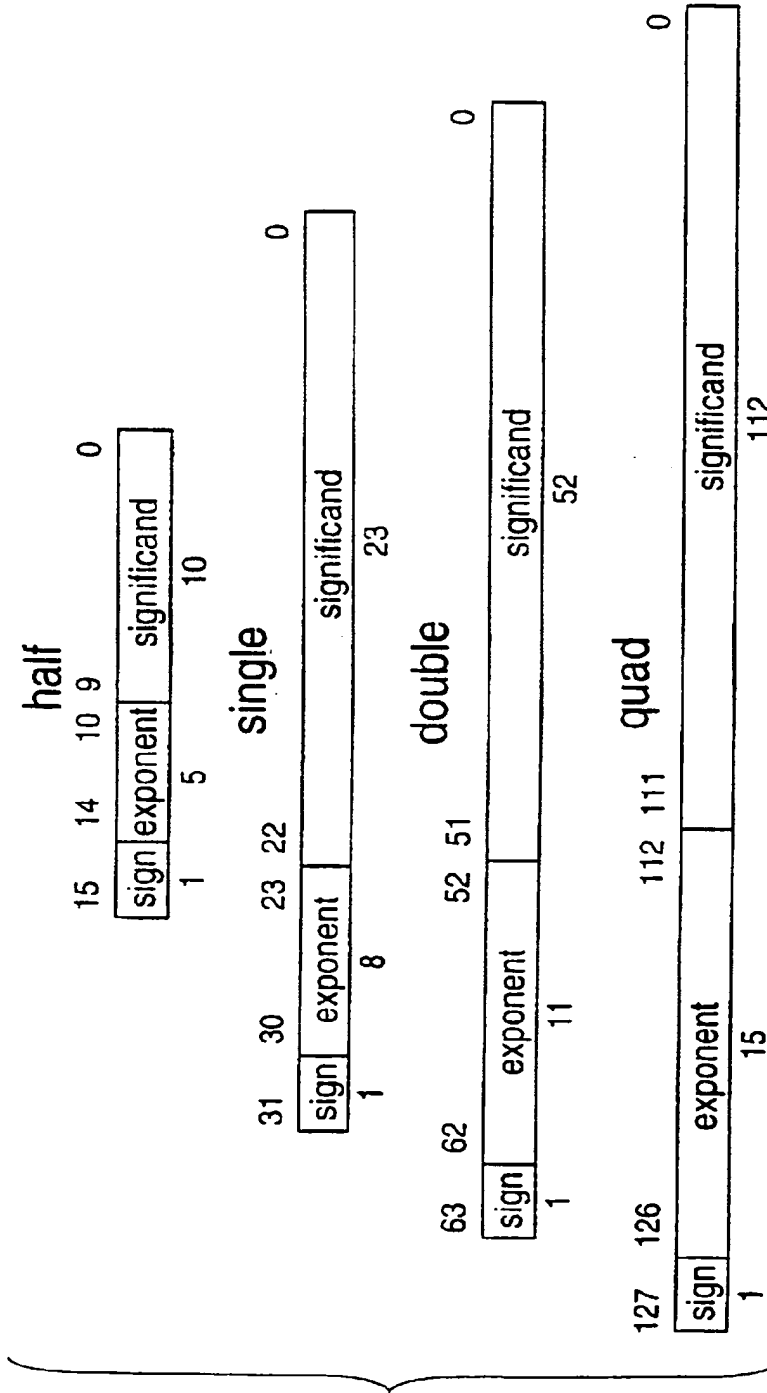
U.S. Patent

Aug. 11, 1998

Sheet 12 of 25

5,794,060

FIG. 9(b)



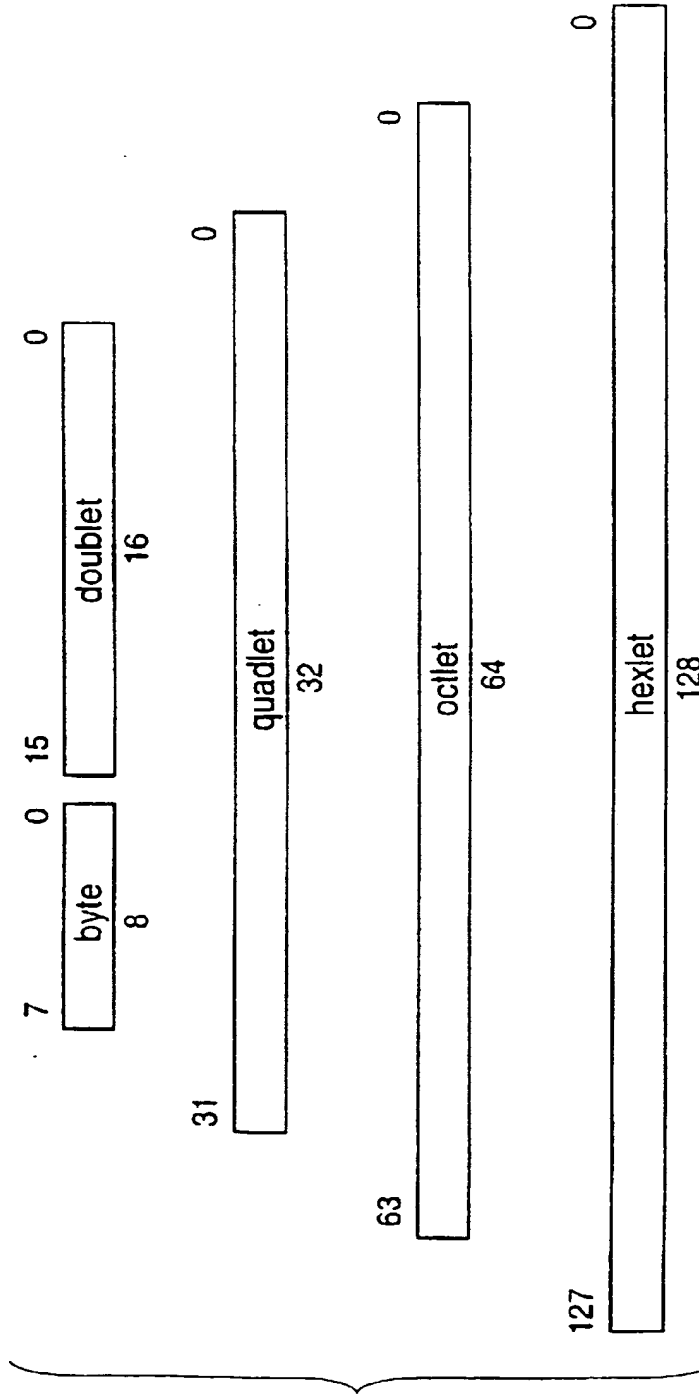
U.S. Patent

Aug. 11, 1998

Sheet 13 of 25

5,794,060

FIG. 9(c)



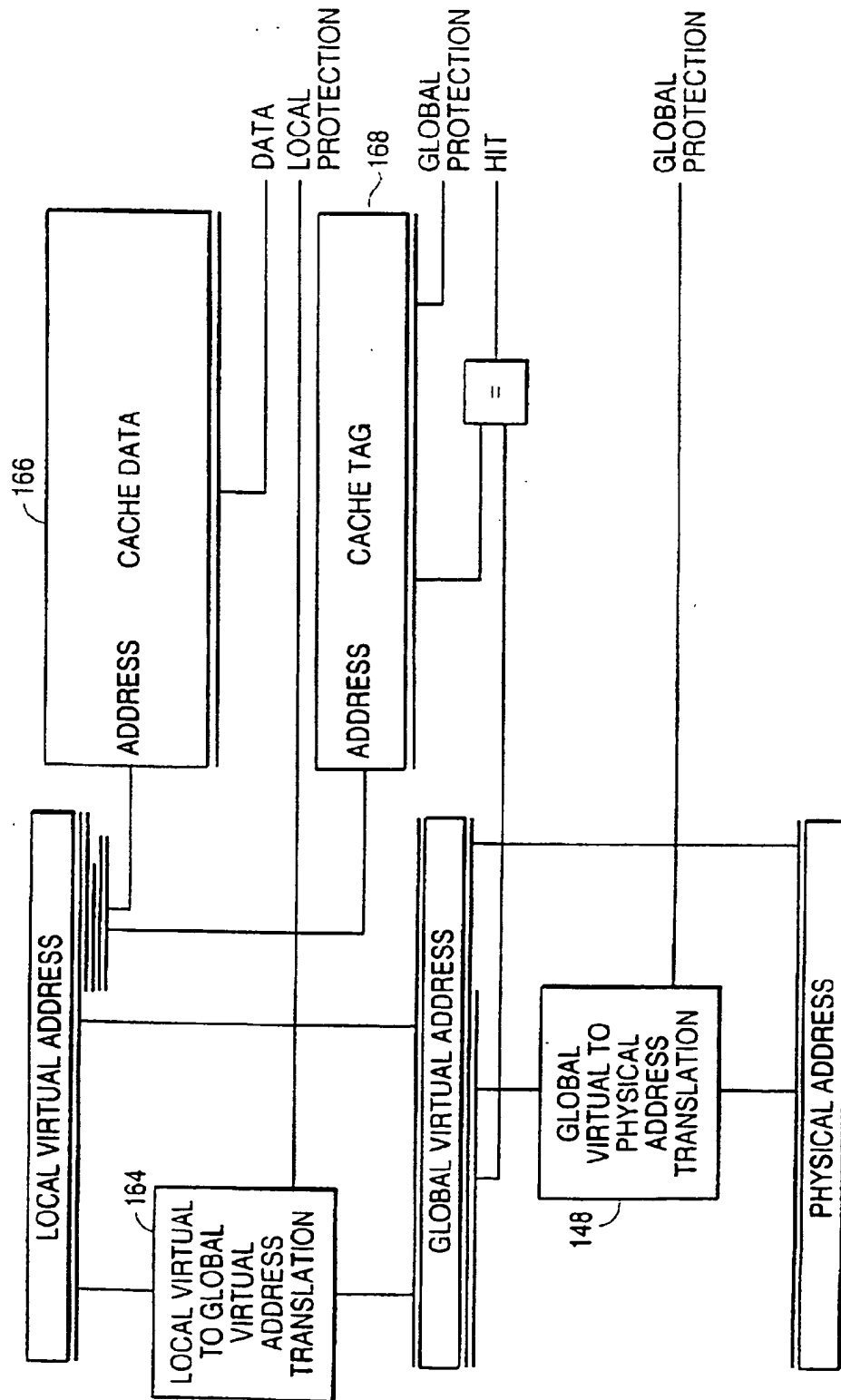
U.S. Patent

Aug. 11, 1998

Sheet 14 of 25

5,794,060

FIG. 10(a)

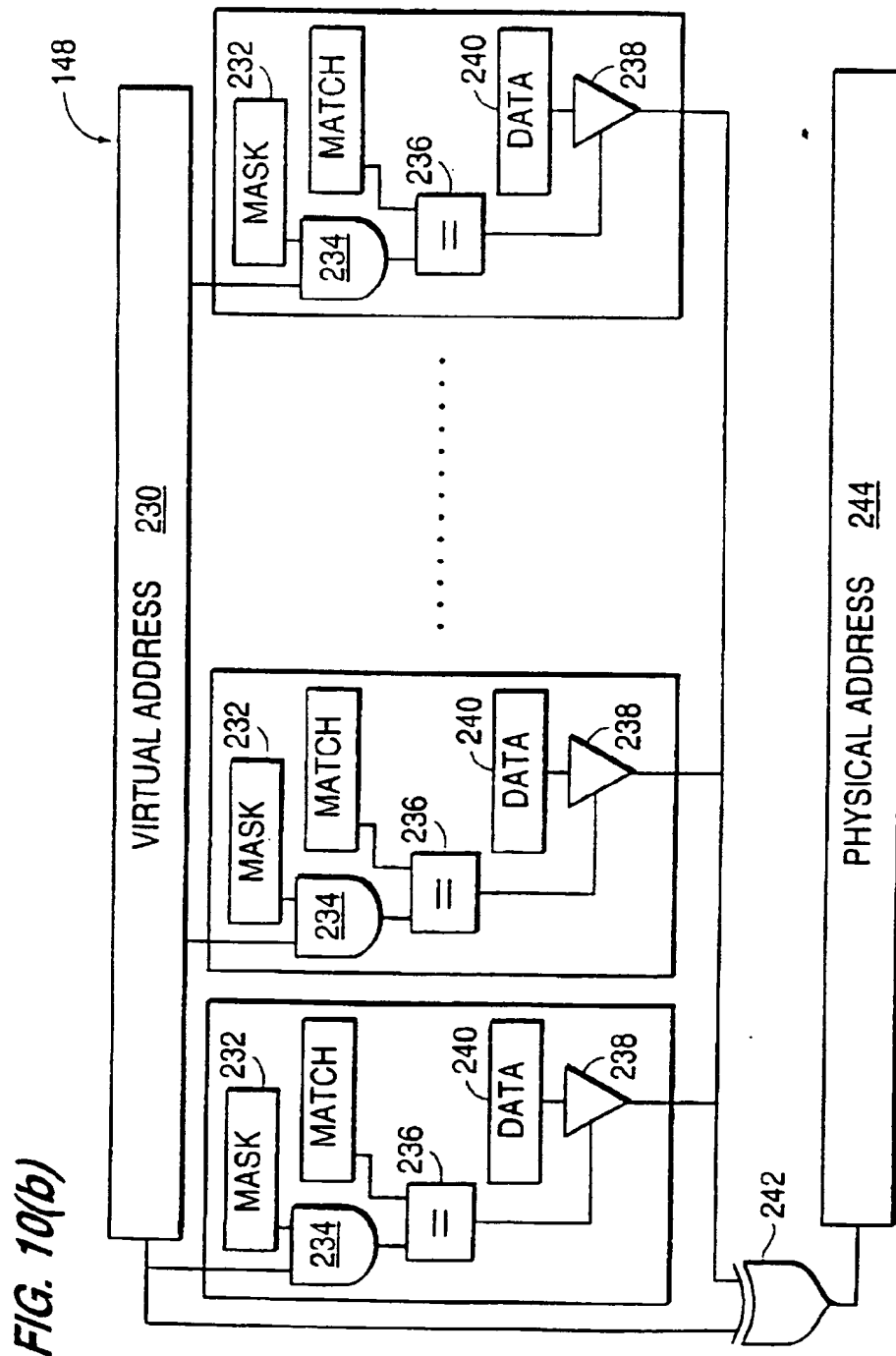


U.S. Patent

Aug. 11, 1998

Sheet 15 of 25

5,794,060



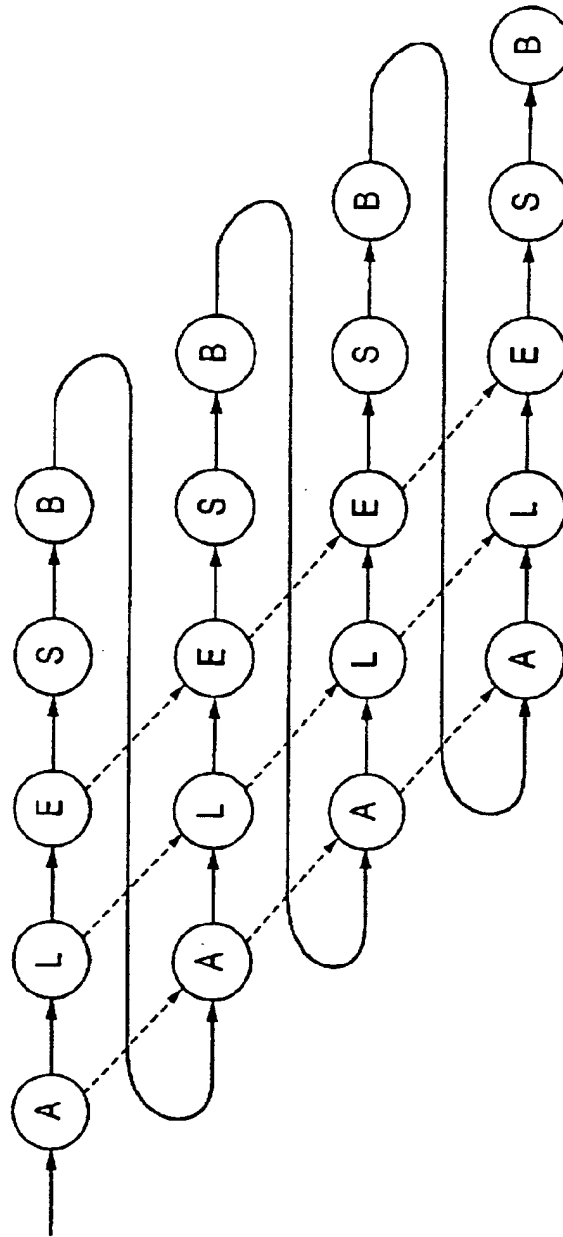
U.S. Patent

Aug. 11, 1998

Sheet 16 of 25

5,794,060

FIG. 11

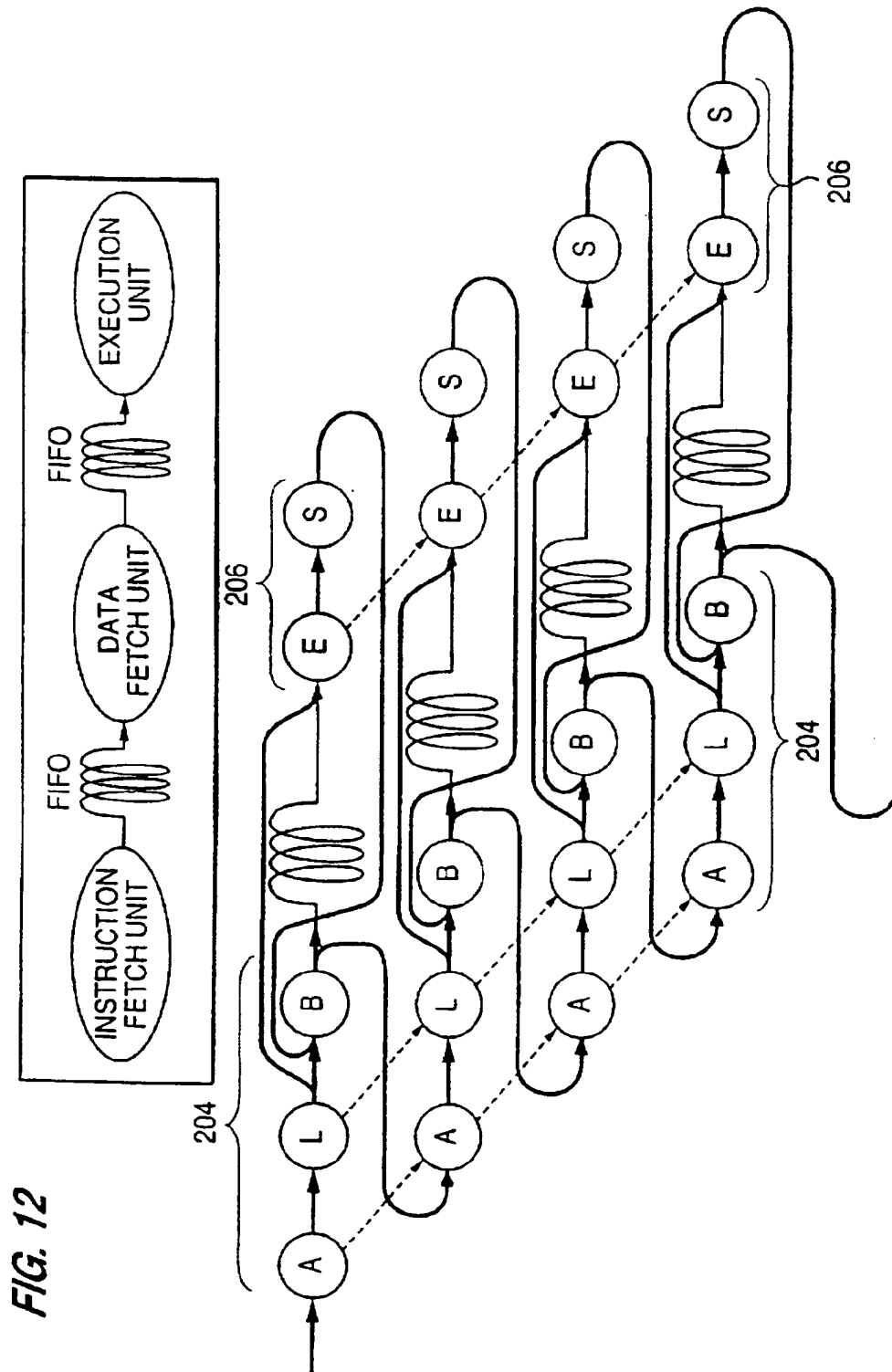


U.S. Patent

Aug. 11, 1998

Sheet 17 of 25

5,794,060



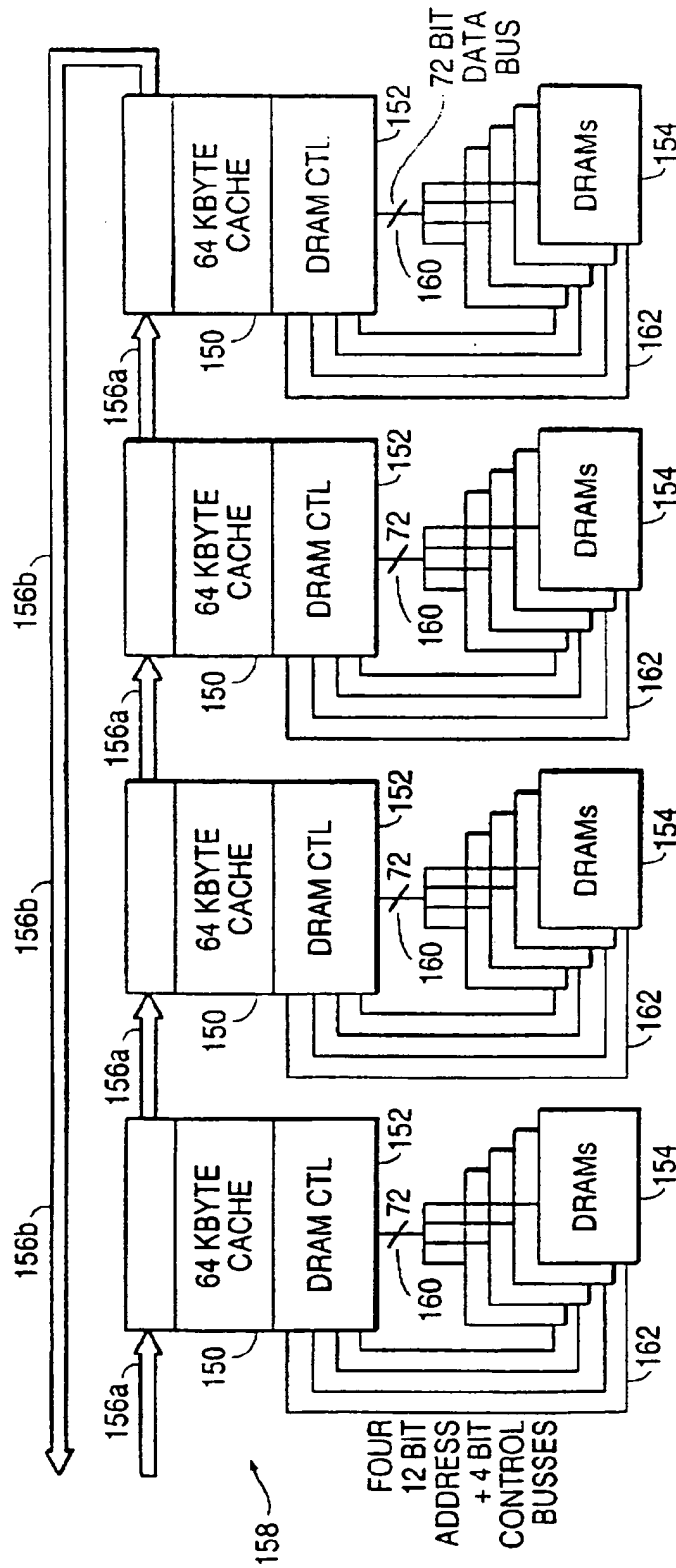
U.S. Patent

Aug. 11, 1998

Sheet 18 of 25

5,794,060

FIG. 13

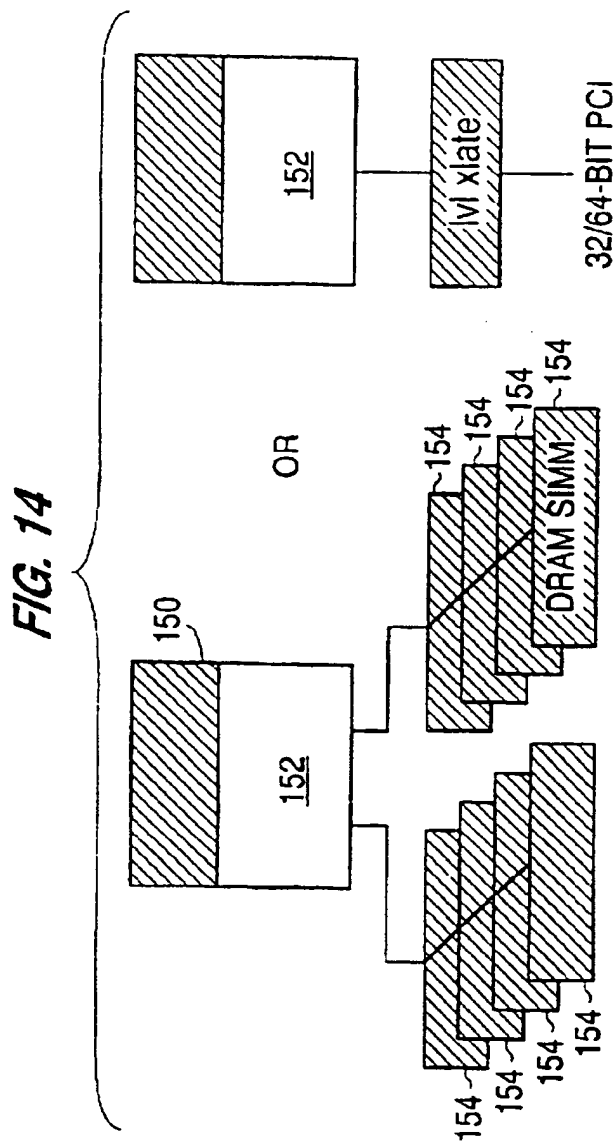


U.S. Patent

Aug. 11, 1998

Sheet 19 of 25

5,794,060



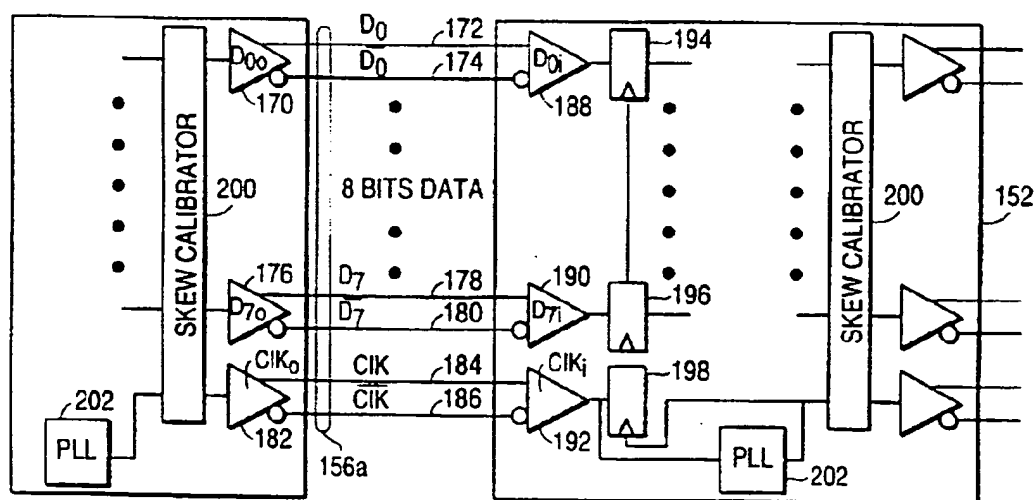
U.S. Patent

Aug. 11, 1998

Sheet 20 of 25

5,794,060

FIG. 15



U.S. Patent

Aug. 11, 1998

Sheet 21 of 25

5,794,060

FIG. 16(a)

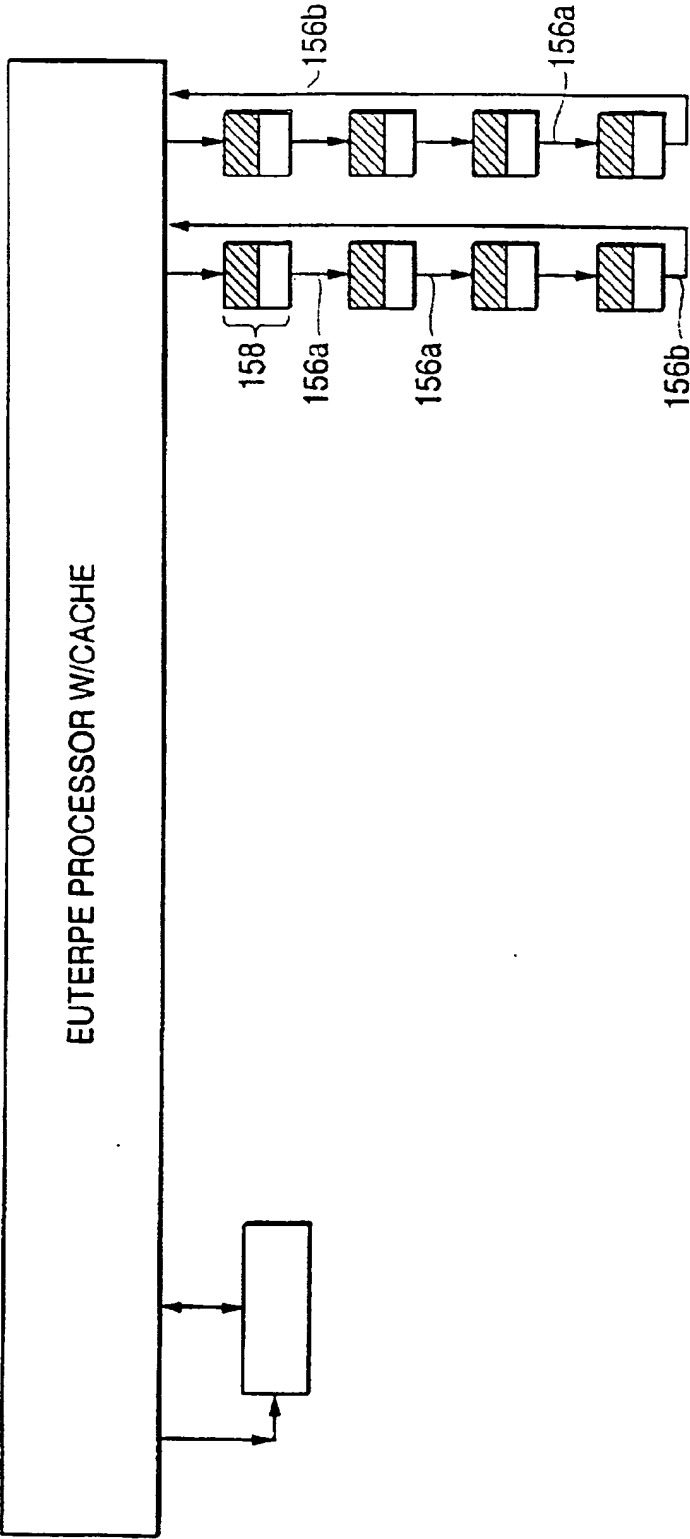
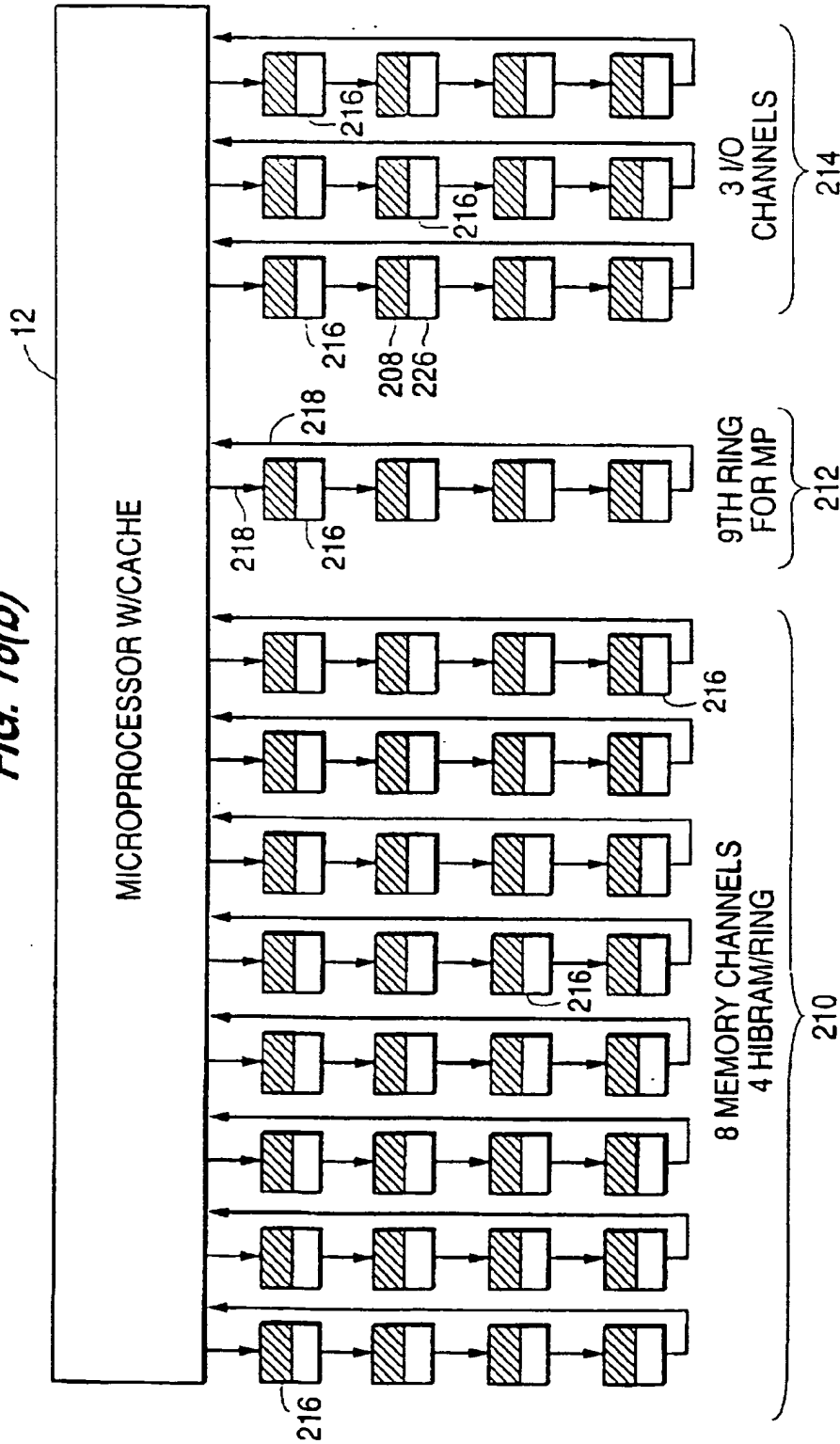


FIG. 16(b)

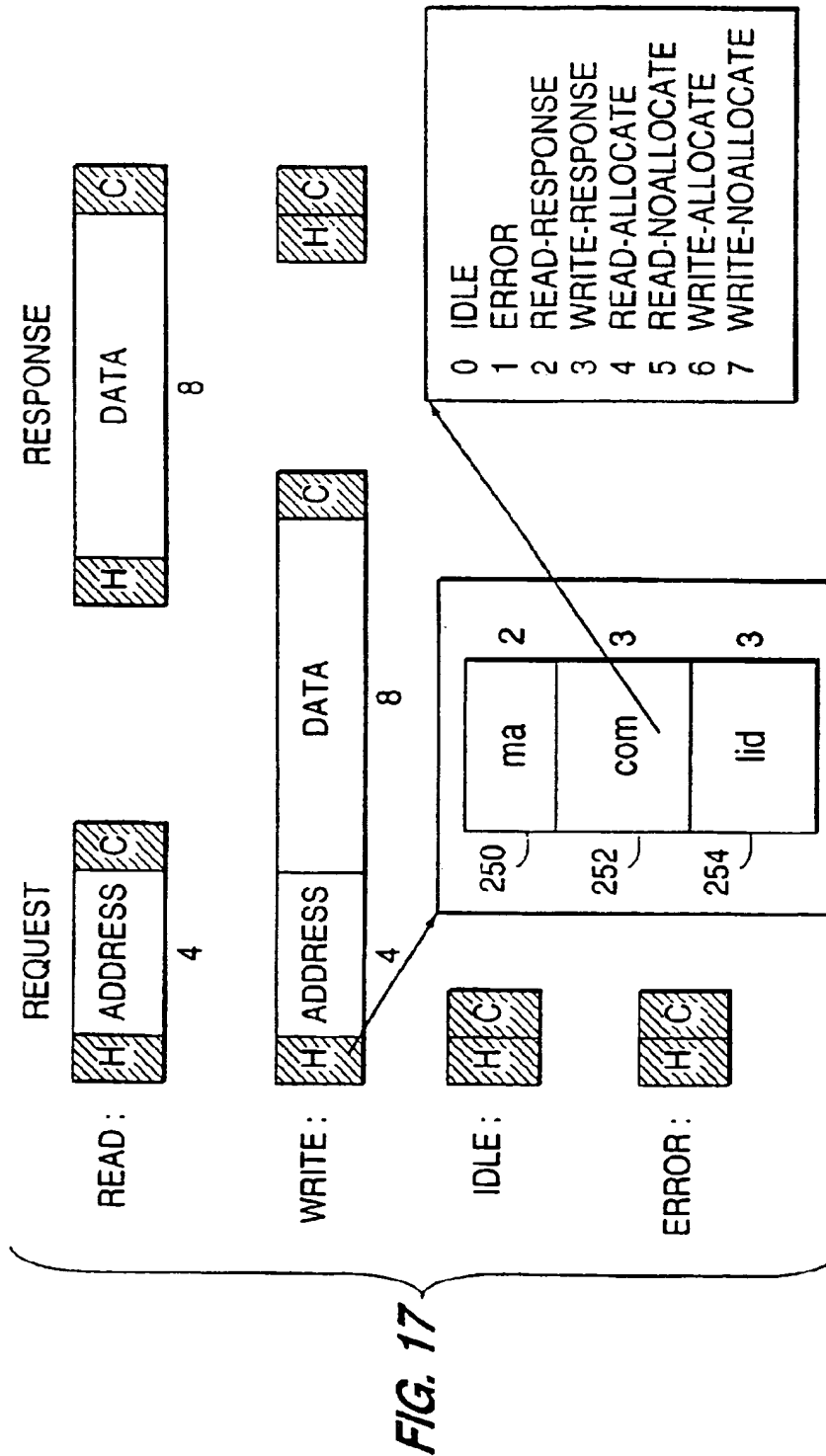


U.S. Patent

Aug. 11, 1998

Sheet 23 of 25

5,794,060



U.S. Patent

Aug. 11, 1998

Sheet 24 of 25

5,794,060

FIG. 18(a)

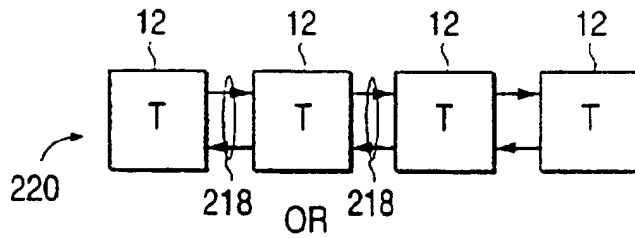


FIG. 18(b)

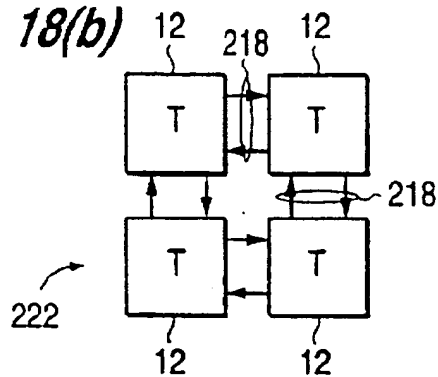
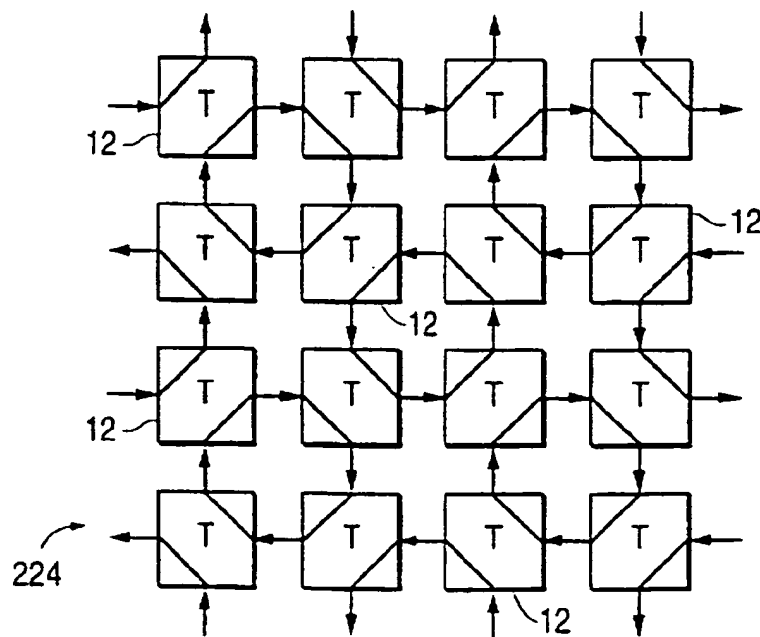


FIG. 18(c)



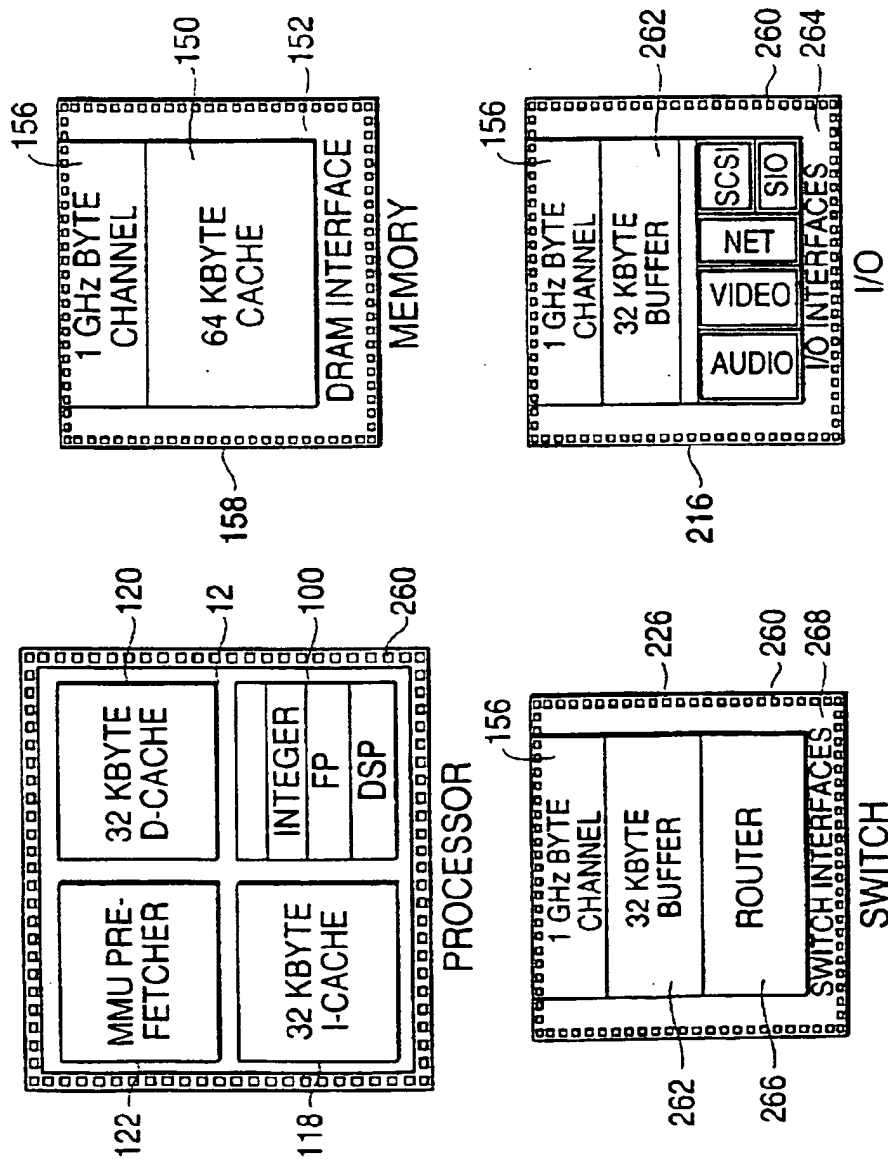
U.S. Patent

Aug. 11, 1998

Sheet 25 of 25

5,794,060

FIG. 19



5,794,060

1

GENERAL PURPOSE, MULTIPLE PRECISION PARALLEL OPERATION, PROGRAMMABLE MEDIA PROCESSOR

This is a divisional of application Ser. No. 08/516,036, filed Aug. 16, 1995, now U.S. Pat. No. 5,742,840.

A Microfiche Appendix consisting of 4 sheets (387 total frames) of microfiche is included in this application. The Microfiche Appendix contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by any one of the Microfiche Appendix, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

FIELD OF THE INVENTION

This invention relates to the field of communications processing, and more particularly, to a method and apparatus for real-time processing of multi-media digital communications.

BACKGROUND OF THE INVENTION

Optical fiber and discs have made the transmission and storage of digital information both cheaper and easier than older analog technologies. An improved system for digital processing of media data streams is necessary in order to realize the full potential of these advanced media.

For the past century, telephone service delivered over copper twisted pair has been the lingua franca of communications. Over the next century, broadband services delivered over optical fiber and coax will more completely fulfill the human need for sensory information by supplying voice, video, and data at rates of about 1,000 times greater than narrow band telephony. Current general-purpose microprocessors and digital signal processors ("DSPs") can handle digital voice, data, and images at narrow band rates, but they are way too slow for processing media data at broadband rates.

This shortfall in digital processing of broadband media is currently being addressed through the design of many different kinds of application-specific integrated circuits ("ASICs"). For example, a prototypical broadband device such as a cable modem modulates and demodulates digital data at rates up to 45 Mbits/sec within a single 6 MHz cable channel (as compared to rates of 28.8 Kbits/sec within a 6 KHz channel for telephone modems) and transcodes it onto a 10/100 base T connection to a personal computer ("PC") or workstation. Current cable modems thus receive data from a coaxial cable connection through a chain of specialized ASIC devices in order to accomplish Quadrature Amplitude Modulation ("QAM") demodulation, Reed-Solomon error correction, packet filtering, Data Encryption Standard ("DES") decryption, and Ethernet protocol handling. The cable modems also transmit data to the coaxial cable link through a second chain of devices to achieve DES encryption, Reed-Solomon block encoding, and Quaternary Phase Shift Keying ("QPSK") modulation. In these environments, a general-purpose processor is usually required as well in order to perform initialization, statistics collection, diagnostics, and network management functions.

The ASIC approach to media processing has three fundamental flaws: cost, complexity, and rigidity. The combined silicon area of all the specialized ASIC devices required in the cable modem, for example, results in a component cost incompatible with the per subscriber price target for a cable service. The cable plant itself is a very

2

hostile service environment, with noise ingress, reflections, nonlinear amplifiers, and other channel impairments, especially when viewed in the upstream direction. Telephony modems have developed an elaborate hierarchy of algorithms implemented in DSP software, with automatic reduction of data rates from 28.8 Kbits/sec to 19.6 Kbits/sec, 14.4 Kbits/sec, or much lower rates as needed to accommodate noise, echoes, and other impairments in the copper plant. To implement similar algorithms on an ASIC-based broadband modem is far more complex to achieve in software.

These problems of cost, complexity, and rigidity are compounded further in more complete broadband devices such as digital set-top boxes, multimedia PCs, or video conferencing equipment, all of which go beyond the basic radio frequency ("RF") modem functions to include a broad range of audio and video compression and decoding algorithms, along with remote control and graphical user interfaces. Software for these devices must control what amounts to a heterogeneous multi-processor, where each specialized processor has a different, and usually eccentric or primitive, programming environment. Even if these programming environments are mastered, the degree of programmability is limited. For example, Motion Picture Expert Group-I ("MPEG-I") chips manufactured by AT&T Corporation will not implement advances such as fractal- and wavelet-based compression algorithms, but these chips are not readily software upgradeable to the MPEG-II standard. A broadband network operator who leases an MPEG ASIC-based product is therefore at risk of having to continuously upgrade his system by purchasing significant amounts of new hardware just to track the evolution of MPEG standards.

The high cost of ASIC-based media processing results from inefficiencies in both memory and logic. A typical ASIC consists of a multiplicity of specialized logic blocks, each with a small memory dedicated to holding the data which comprises the working set for that block. The silicon area of these multiple small memories is further increased by the overhead of multiple decoders, sense amplifiers, write drivers, etc. required for each logic block. The logic blocks are also constrained to operate at frequencies determined by the internal symbol rates of broadband algorithms in order to avoid additional buffer memories. These frequencies typically differ from the optimum speed-area operating point of a given semiconductor technology. Interconnect and synchronization of the many logic and memory blocks are also major sources of overhead in the ASIC approach.

The disadvantages of the prior ASIC approach can be overcome by a single unified media processor. The cost advantages of such a unified processor can be achieved by gathering all the many ASIC functions of a broadband media product into a single integrated circuit. Cost reduction is further increased by reducing the total memory area of such a circuit by replacing the multiplicity of small ASIC memories with a single memory hierarchy large enough to accommodate the sum total of all the working sets, and wide enough to supply the aggregate bandwidth needs of all the logic blocks. Additionally, the logic block interconnect circuitry to this memory hierarchy may be streamlined by providing a generally programmable switching fabric. Many of the logic blocks themselves can also be replaced with a single multi-precision arithmetic unit, which can be internally partitioned under software control to perform addition, multiplication, division, and other integer and floating point arithmetic operations on symbol streams of varying widths, while sustaining the full data throughput of the memory hierarchy. The residue of logic blocks that perform opera-

5,794,060

3

tions that are neither arithmetic or permutation group oriented can be replaced with an extended math unit that supports additional arithmetic operations such as finite field, ring, and table lookup, while also sustaining the full data throughput of the memory hierarchy.

The above multi-precision arithmetic, permutation switch, and extended math operations can then be organized as machine instructions that transfer their operands to and from a single wide multi-ported register file. These instructions can be further supplemented with load/store instructions that transfer register data to and from a data buffer/cache static random access memory ("SRAM") and main memory dynamic random access memories ("DRAMs"), and with branch instructions that control the flow of instructions executed from an instruction buffer/cache SRAM. Extensions to the load/store instructions can be made for synchronization, and to branch instructions for protected gateways, so that multiple threads of execution for audio, video, radio, encryption, networking, etc. can efficiently and securely share memory and logic resources of a unified machine operating near the optimum speed-area point of the target semiconductor process. The data path for such a unified media processor can interface to a high speed input/output ("I/O") subsystem that moves media streams across ultra-high bandwidth interfaces to external storage and I/O.

Such a device would incorporate all of the processing capabilities of the specialized multi-ASIC combination into a single, unified processing device. The unified processor would be agile and capable of reprogramming through the transmission of new programs over the communication medium. This programmable, general purpose device is thus less costly than the specialized processor combination, easier to operate and reprogram and can be installed or applied in many differing devices and situations. The device may also be scalable to communications applications that support vast numbers of users through massively parallel distributed computing.

It is therefore an object of this invention to process media data streams by executing operations at very high bandwidth rates.

It is also an object of this invention to unify the audio, video, radio, graphics, encryption, authentication, and networking protocols into a single instruction stream.

It is also an object of this invention to achieve high bandwidth rates in a unified processor that is easy to program and more flexible than a heterogeneous combination of special purpose processors.

It is a further object of the invention to support high level mathematical processing in a unified media processor, including finite group, finite field, finite ring and table look-up operations, all at high bandwidth rates.

It is yet a further object of the invention to provide a unified media processor that can be replicated into a multi-processor system to support a vast array of users.

It is yet another object of this invention to allow for massively parallel systems within the switching fabric to support very large numbers of subscribers and services.

It is also an object of the invention to provide a general purpose programmable processor that could be employed at all points in a network.

It is a further object of this invention to sustain very high bandwidth rates to arbitrarily large memory and input/output systems.

SUMMARY OF THE INVENTION

In view of the above, there is provided a system for media processing that maintains substantially peak data throughput

4

in the execution and transmission of multiple media data streams. The system includes in one aspect a general purpose, programmable media processor, and in another aspect includes a method for receiving, processing and transmitting media data streams. The general purpose, programmable media processor of the invention further includes an execution unit, high bandwidth external interface, and can be employed in a parallel multi-processor system.

According to the apparatus of the invention, an execution unit is provided that maintains substantially peak data throughput in the unified execution of multiple media data streams. The execution unit includes a data path, and a multi-precision arithmetic unit coupled to the data path and capable of dynamic partitioning based on the elemental width of data received from the data path. The execution unit also includes a switch coupled to the data path that is programmable to manipulate data received from the data path and provide data streams to the data path. An extended mathematical element is also provided, which is coupled to the data path and programmable to implement additional mathematical operations at substantially peak data throughput. In a preferred embodiment of the execution unit, at least one register file is coupled to the data path.

According to another aspect of the invention, a general purpose programmable media processor is provided having an instruction path and a data path to digitally process a plurality of media data streams. The media processor includes a high bandwidth external interface operable to receive a plurality of data of various sizes from an external source and communicate the received data over the data path at a rate that maintains substantially peak operation of the media processor. At least one register file is included, which is configurable to receive and store data from the data path and to communicate the stored data to the data path. A multi-precision execution unit is coupled to the data path and is dynamically configurable to partition data received from the data path to account for the elemental symbol size of the plurality of media streams, and is programmable to operate on the data to generate a unified symbol output to the data path.

According to the preferred embodiment of the media processor, means are included for moving data between registers and memory by performing load and store operations, and for coordinating the sharing of data among a plurality of tasks by performing synchronization operations based upon instructions and data received by the execution unit. Means are also provided for securely controlling the sequence of execution by performing branch and gateway operations based upon instructions and data received by the execution unit. A memory management unit operable to retrieve data and instructions for timely and secure communication over the data path and instruction path respectively is also preferably included in the media processor. The preferred embodiment also includes a combined instruction cache and buffer that is dynamically allocated between cache space and buffer space to ensure real-time execution of multiple media instruction streams, and a combined data cache and buffer that is dynamically allocated between cache space and buffer space to ensure real-time response for multiple media data streams.

In another aspect of the invention, a high bandwidth processor interface for receiving and transmitting a media stream is provided having a data path operable to transmit media information at sustained peak rates. The high bandwidth processor interface includes a plurality of memory controllers coupled in series to communicate stored media

5.794.060

5

information to and from the data path, and a plurality of memory elements coupled in parallel to each of the plurality of memory controllers for storing and retrieving the media information. In the preferred embodiment of the high bandwidth processor interface, the plurality of memory controllers each comprise a paired link disposed between each memory controller, where the paired links each transmit and receive plural bits of data and have differential data inputs and outputs and a differential clock signal.

Yet another aspect of the invention includes a system for unified media processing having a plurality of general purpose media processors, where each media processor is operable at substantially peak data rates and has a dynamically partitioned execution unit and a high bandwidth interface for communicating to memory and input/output elements to supply data to the media processor at substantially peak rates. A bi-directional communication fabric is provided, to which the plurality of media processors are coupled, to transmit and receive at least one media stream comprising presentation, transmission, and storage media information. The bi-directional communication fabric preferably comprises a fiber optic network, and a subset of the plurality of media processors comprise network servers.

According to yet another aspect of the invention, a parallel multi-media processor system is provided having a data path and a high bandwidth external interface coupled to the data path and operable to receive a plurality of data of various sizes from an external source and communicate the received data at a rate that maintains substantially peak operation of the parallel multi-processor system. A plurality of register files, each having at least one register coupled to the data path and operable to store data, are also included. At least one multi-precision execution unit is coupled to the data path and is dynamically configurable to partition data received from the data path to account for the elemental symbol size of the plurality of media streams, and is programmable to operate in parallel on data stored in the plurality of register files to generate a unified symbol output for each register file.

According to the method of the invention, unified streams of media data are processed by receiving a stream of unified media data including presentation, transmission and storage information. The unified stream of media data is dynamically partitioned into component fields of at least one bit based on the elemental symbol size of data received. The unified stream of media data is then processed at substantially peak operation.

In one aspect of the invention, the unified stream of media data is processed by storing the stream of unified media data in a general register file. Multi-precision arithmetic operations can then be performed on the stored stream of unified media data based on programmed instructions, where the multi-precision arithmetic operations include Boolean, integer and floating point mathematical operations. The component fields of unified media data can then be manipulated based on programmed instructions that implement copying, shifting and re-sizing operations. Multi-precision mathematical operations can also be performed on the stored stream of unified media data based on programmed instructions, where the mathematical operations including finite group, finite field, finite ring and table look-up operations. Instruction and data pre-fetching are included to fill instruction and data pipelines, and memory management operations can be performed to retrieve instructions and data from external memory. The instructions and data are preferably stored in instruction and data cache/buffers, in which buffer storage in the instruction and data cache/buffers is dynamically allocated to ensure real-time execution.

6

Other aspects of the invention include a method for achieving high bandwidth communications between a general purpose media processor and external devices by providing a high bandwidth interface disposed between the media processor and the external devices, in which the high bandwidth interface comprises at least one uni-directional channel pair having an input port and an output port. A plurality of media data streams, comprising component fields of various sizes, are transmitted and received between the media processor and the external devices at a rate that sustains substantially peak data throughput at the media processor. A method for processing streams of media data is also included that provides a bi-directional communications fabric for transmitting and receiving at least one stream of media data, where the at least one stream of media data comprises presentation, transmission and storage information. At least one programmable media processor is provided within the communications network for receiving, processing and transmitting the at least one stream of unified media data over the bi-directional communications fabric.

The general purpose, programmable media processor of the invention combines in a single device all of the necessary hardware included in the specialized processor combinations to process and communicate digital media data streams in real-time. The general purpose, programmable media processor is therefore cheaper and more flexible than the prior approach to media processing. The general purpose, programmable media processor is thus more susceptible to incorporation within a massively parallel processing network of general purpose media processors that enhance the ability to provide real-time multi-media communications to the masses.

These features are accomplished by deploying server media processors and client media processors throughout the network. Such a network provides a seamless, global media super-computer which allows programmers and network owners to virtualize resources. Rather than restrictively accessing only the memory space and processing time of a local resource, the system allows access to resources throughout the network. In small access points such as wireless devices, where very little memory and processing logic is available due to limited battery life, the system is able to draw upon the resources of a homogeneous multi-computer system.

The invention also allows network owners the facility to track standards and to deploy new services by broadcasting software across the network rather than by instituting costly hardware upgrades across the whole network. Broadcasting software across the network can be performed at the end of an advertisement or other program that is broadcasted nationally. Thus, services can be advertised and then transmitted to new subscribers at the end of the advertisement.

These and other features and advantages of the invention will be apparent upon consideration of the following detailed description of the presently preferred embodiments of the invention, taken in conjunction with the appended drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a broad band media computer employing the general purpose, programmable media processor of the invention;

FIG. 2 is a block diagram of a global media processor employing multiple general purpose media processors according to the invention;

FIG. 3 is an illustration of the digital bandwidth spectrum for telecommunications, media and computing communications;

5,794,060

7

FIG. 4 is the digital bandwidth spectrum shown in FIG. 3 taking into account the bandwidth overhead associated with compressed video techniques;

FIG. 5 is a block diagram of the current specialized processor solution for mass media communication, where FIG. 5 shows the current distributed system, and shows a possible integrated approach;

FIG. 6 is a block diagram of two presently preferred general purpose media processors, where FIG. 6 shows a distributed system and shows an integrated media processor;

FIG. 7 is a block diagram of the presently preferred structure of a general purpose, programmable media processor according to the invention;

FIG. 8 is a drawing consisting of visual illustrations of the various group operations provided on the media processor, where FIG. 8(a) illustrates the group expand operation, FIG. 8(b) illustrates the group compress or extract operation, FIG. 8(c) illustrates the group deal and shuffle operations, FIG. 8(d) illustrates the group swizzle operation and FIG. 8(e) illustrates the various group permute operations;

FIG. 9 shows the preferred instruction and data sizes for the general purpose, programmable media processor, where FIG. 9(a) is an illustration of the various instruction formats available on the general purpose, programmable media processor, FIG. 9(b) illustrates the various floating-point data sizes available on the general purpose media processor, and FIG. 9(c) illustrates the various fixed-point data sizes available on the general purpose media processor;

FIG. 10 is an illustration of a presently preferred memory management unit included in the general purpose processor shown in FIG. 7, where FIG. 10(a) is a translation block diagram and FIG. 10(b) illustrates the functional blocks of the transaction lookaside buffer;

FIG. 11 is an illustration of a super-string pipeline technique;

FIG. 12 is an illustration of the presently preferred super-string pipeline technique;

FIG. 13 is a block diagram of a single memory channel for communication to the general purpose media processor shown in FIG. 7;

FIG. 14 is an illustration of the presently preferred connection of standard memory devices to the preferred memory interface;

FIG. 15 is a block diagram of the input/output controller for use with the memory channel shown in FIG. 13;

FIG. 16 is a block diagram showing multiple memory channels connected to the general purpose media processor shown in FIG. 7, where FIG. 16(a) shows a two-channel implementation and FIG. 16(b) illustrates a twelve-channel embodiment;

FIG. 17 illustrates the presently preferred packet communications protocol for use over the memory channel shown in FIG. 13;

FIG. 18 shows a multi-processor configuration employing the general purpose media processor shown in FIG. 7, where FIG. 18(a) shows a linear processor configuration, FIG. 18(b) shows a processor ring configuration, and FIG. 18(c) shows a two-dimensional processor configuration; and

FIG. 19 shows a presently preferred multi-chip implementation of the general purpose, programmable media processor of the invention.

DETAILED DESCRIPTION OF THE PRESENTLY PREFERRED EMBODIMENTS

Referring to the drawings, where like-reference numerals refer to like elements throughout, a broad band microcom-

8

puter 10 is provided in FIG. 1. The broad band microcomputer 10 consists essentially of a general purpose media processor 12. As will be described in more detail below, the general purpose media processor 12 receives, processes and transmits media data streams in a bi-directional manner from upstream network components to downstream devices. In general, media data streams received from upstream network components can comprise any combination of audio, video, radio, graphics, encryption, authentication, and networking information. As those skilled in the art will appreciate, however, the general purpose media processor 12 is in no way limited to receiving, processing and transmitting only these types of media information. The general purpose media processor 12 of the invention is capable of processing any form of digital media information without departing from the spirit and essential scope of the invention.

System Configuration

In the preferred embodiment of the invention shown in FIG. 1, media data streams are communicated to the media processor 12 from several sources. Ideally, unified media data streams are received and transmitted by the general purpose media processor 12 over a fiber optic cable network 14. As will be described in more detail below, although a fiber optic cable network is preferred, the presently existing communications network in the United States consists of a combination of fiber optic cable, coaxial cable and other transmission media. Consequently, the general purpose media processor 12 can also receive and transmit media data streams over coaxial cable 14 and traditional twisted pair wire connections 16. The specific communications protocol employed over the twisted pair 16, whether POTS, ISDN or ADSL, is not essential; all protocols are supported by the broad band microcomputer 10. The details of these protocols are generally known to those skilled in the art and no further discussion is therefore needed or provided herein.

Another form of upstream network communication is through a satellite link 18. The satellite link 18 is typically connected to a satellite receiver 20. The satellite receiver 20 comprises an antenna, usually in the form of a satellite dish, and amplification circuitry. The details of such satellite communications are also generally known in the art, and further detail is therefore not provided or included herein.

As described above, the general purpose media processor 12 communicates in a bi-directional manner to receive, process and transmit media data streams to and from downstream devices. As shown in FIG. 1, downstream communication preferably takes place in at least two forms. First, media data streams can be communicated over a bi-directional local network 22. Various types of local networks 22 are generally known in the art and many different forms exist. The general purpose media processor 12 is capable of communicating over any of these local networks 22 and the particular type of network selected is implementation specific.

The local network 22 is preferably employed to communicate between the unified processor 12, and audio/visual devices 24 or other digital devices 26. Presently preferred examples of audio/visual devices 24 include digital cable television, video-on-demand devices, electronic yellow pages services, integrated message systems, video telephones, video games and electronic program guides. As those skilled in the art will appreciate, other forms of audio/video devices are contemplated within the spirit and scope of the invention. Presently preferred embodiments of

5,794,060

9

other digital devices 26 for communication with the general purpose media processor 12 include personal computers, television sets, work stations, digital video camera recorders, and compact disc read-only memories. As those skilled in the art will also appreciate, further digital devices 26 are contemplated for communication to the general purpose media processor 12 without departing from the spirit and scope of the invention.

Second, the general purpose media processor preferably also communicates with downstream devices over a wireless network 28. In the presently preferred embodiment of the invention, wireless devices for communication over the wireless network 28 can comprise either remote communication devices 30 or remote computing devices 32. Presently preferred embodiments of the remote communications devices 30 include cordless telephones and personal communicators. Presently preferred embodiments of the remote computing devices 32 include remote controls and telecommunicating devices. As those skilled in the art will appreciate, other forms of remote communication devices 30 and remote computing devices 32 are capable of communication with the general purpose media processor 12 without departing from the spirit and scope of the invention. An agile digital radio (not shown) that incorporates a general purpose media processor 12 may be used to communicate with these wireless devices.

Network Configuration

Referring now to FIG. 2, the general purpose media processor 12 is preferably disposed throughout a digital communications network 38. In order to enable communication among large and small businesses, residential customers and mobile users, the network 38 can consist of a combination of many individual sub-networks comprised of three main forms of interconnection. The trunk and main branches of the network 38 preferably employ fiber optic cable 40 as the preferred means of interconnection. Fiber optic cable 40 is used to connect between general purpose media processors 12 disposed as network servers 46 or large business installations 48 that are capable of coupling directly to the fiber optic link 40. For communications to small business and residential customers that may be incapable of directly coupling to the fiber optic cable 40, a general purpose media processor 12 can be used as an interface to other forms of network interconnection.

As shown in FIG. 2, alternate forms of interconnection consist of coaxial cable lines 42 and twisted pair wiring 44. Coaxial cable lines are currently in place throughout the U.S. and is typically employed to provide cable television services to residential homes. According to the preferred embodiment of the invention, general purpose media processors 12 can be installed at these residential locations 52. In contrast to the specialized processor approach, the general purpose media processor 12 provides enough bandwidth to allow for bi-directional communications to and from these residential locations 52.

Network servers 46 controlled by general purpose media processors 12 are also employed throughout the network 38. For example, the network servers 46 can be used to interface between the fiber optic network 40 and twisted pair wiring 44. Twisted pair wiring 44 is still employed for small businesses 50 and residential locations 52 that do not or cannot currently subscribe to coaxial cable or fiber optic network services. General purpose media processors 12 are also disposed at these small business locations 50 and non-cable residential locations 52. General purpose media

10

processors 12 are also installed in wireless or mobile locations 52, which are coupled to the network 38 through agile digital radios (not shown). As shown in FIG. 2, network databases or other peripherals 56 can also coupled to general purpose media processors 12 in the network 38.

The general purpose media processor 12 is operable at significantly high bandwidths in order to receive, process and transmit unified media data streams. Referring to FIG. 3, the respective frequencies for various types of media data streams are set forth against a bandwidth spectrum 60. The bandwidth spectrum 60 includes three component spectrums, all along the same range of frequencies, which represent the various frequency rates of digital media communications. Current computing bandwidth capabilities are also displayed. The telecommunications spectrum 62 shows the various frequency bands used for telecommunications transmission. For example, teletype terminals and modems operate in a range between approximately 64 bits/second to 16 kilobits/second. The ISDN telecommunication protocol operates at 64 kilobits/second. At the upper end of the telecommunications spectrum 62, T1 and T3 trunks operate at one megabit per second and 32 megabits per second, respectively. The SONET frequency range extends from approximately 128 megabits per second up to approximately 32 gigabits per second. Accordingly, in order to carry such broad band communications, the general purpose media processor 12 is capable of transferring information at rates into the gigabits per second range or higher.

A spectrum of typical media data streams is presented in the media spectrum 64 shown in FIG. 3. Voice and music transmissions are centered at frequencies of approximately 64 kilobits per second and one megabit per second, respectively. At the upper end of the media spectrum 64, video transmission takes place in a range from 128 megabits per second for high density television up to over 256 gigabits per second for movie applications. When using common video compression techniques, however, the video transmission spectrum can be shifted down to between 32 kilobits per second to 128 megabits per second as a result of the data compression. As described below, the processing required to achieve the data compression results in an increase in bandwidth requirements.

Current computing bandwidths are shown in the computing spectrum 66 of FIG. 3. Serial communications presently take place in a range between two kilobits per second up to 512 kilobits per second. The Ethernet network protocol operates at approximately 8 megabits per second. Current dynamic random access memory and other digital input/output peripherals operate between 32 megabits per second and 512 megabits per second. Presently available microprocessors are capable of operation in the low gigabits per second range. For example, the '386 Pentium microprocessor manufactured by Intel Corporation of Santa Clara, Calif. operates in the lower half of that range, and the Alpha microprocessor manufactured by Digital Equipment Corporation approaches the 16 gigabits per second range.

When video compression is employed, as expressed above, the associated processing overhead reduces the effective bandwidth of the particular processor. As a result, in order to handle compressed video, these processors must operate in the terahertz frequency range. The bandwidth spectrum 60 shown in FIG. 4 represents the effect of handling media data streams including compressed video. The computing spectrum 66 is skewed down to properly align the computing bandwidth requirements with the telecommunications spectrum 62 and the media spectrum 64. Accordingly, current processor technology is not sufficient

5.794.060

11

to handle the transmission and processing associated with complex streams of multi-media data.

The current specialized processor approach to media processing is illustrated in the block diagram shown in FIG. 5. As shown in FIG. 5, special purpose processors are coupled to a back plane 70, which is capable of transmitting instructions and data at the upper kilobits to lower gigabits per second range. In a typical configuration, an audio processor 76, video processor 78, graphics processor 80 and network processor 82 are all coupled to the back plane 70. Each of the audio, video, graphics and network processors 76-82 typically employ their own private or dedicated memories 84, which are only accessible to the specific processor and not accessible over the back plane 70. As described above, however, unless video data streams are constantly being processed, for example, the video processor 78 will sit idle for periods of time. The computing power of the dedicated video processor 78 is thus only available to handle video data streams and is not available to handle other media data streams that are directed to other dedicated processors. This, of course, is an inefficient use of the video processor 78 particularly in view of the overall processing capability of this multi-processor system.

The general purpose media processor 12, in contrast, handles a data stream of audio, video, graphics and network information all at the same time with the same processor. In order to handle the ever changing combination of data types, the general purpose media processor 12 is dynamically partitionable to allocate the appropriate amount of processing for each combination of media in a unified media data stream. A block diagram of two preferred general purpose media processor system configurations is shown in FIG. 6. Referring to FIG. 6, a general purpose media processor 12 is coupled to a high-speed back plane 90. The presently preferred back plane 90 is capable of operation at 30 gigabits per second. As those skilled in the art will appreciate, back planes 90 that are capable of operation at 400 gigabits per second or greater bandwidth are envisioned within the spirit and scope of the invention. Multiple memory devices 92 are also coupled to the back plane 90, which are accessible by the general purpose media processor 12. Input/output devices 94 are coupled to the back plane 90 through a dual-ported memory 92. The configuration of the input/output devices 94 on one end of the dual-ported memory 92 allows the sharing of these memory devices 92 throughout a network 38 of general purpose media processors 12.

Alternatively, FIG. 6 shows a presently preferred integrated general purpose media processor 12. The integrated processor includes on-board memory and I/O 86. The on-board memory is preferably of sufficient size to optimize throughput, and can comprise a cache and/or buffer memory or the like. The integrated media processor 12 also connects to external memory 86, which is preferably larger than the on-board memory 86 and forms the system main memory.

Execution Unit

One presently preferred embodiment of an integrated general purpose media processor 12 is shown in FIG. 7. The core of the integrated general purpose media processor 12 comprises an execution unit 100. Three main elements or subsections are included in the execution unit 100. A multiple precision arithmetic/logic unit ("ALU") 102 performs all logical and simple arithmetic operations on incoming media data streams. Such operations consist of calculate and control operations such as Boolean functions, as well as addition, subtraction, multiplication and division. These

12

operations are performed on single or unified media data streams transmitted to and from the multiple precision ALU 102 over a data bus or data path 108. Preferably the data path 108 is 128 bits wide, although those skilled in the art will appreciate that the data path 108 can take on any width or size without departing from the spirit and scope of the invention. The wider the data path 108 the more unified media data can be processed in parallel by the general purpose media processor 12.

Coupled to the multi-precision ALU 102 via the data path 108, and also an element of the execution unit 100, is a programmable switch 104. The programmable switch 104 performs data handling operations on single or unified media data streams transmitted over the data path 108. Examples of such data handling operations include deals, shuffles, shifts, expands, compresses, swizzles, permutes and reverses, although other data handling operations are contemplated. These operations can be performed on single bits or bit fields consisting of two or more bits up to the entire width of the data path 108. Thus, single bits or bit fields of various sizes can be manipulated through programmable operation of the switch 104.

Examples of the presently preferred data manipulation operations performed by the general purpose media processor 12 are shown in FIG. 8. A group expand operation is visually illustrated in FIG. 8(a). According to the group expand operation, a sequential field of bits 270 can be divided into constituent sub-fields 272a-272d for insertion into a larger field array 274. The reverse of the group expand operation is a group compress or extract operation. A visual illustration of the group compress or extract operation is shown in FIG. 8(b). As shown, separate sub-fields 272a-272d from a larger bit field 274 can be combined to form a contiguous or sequential field of bits 270.

Referring to FIGS. 8(c)-8(e), group deal, shuffle, swizzle and permute operations performed by the programmable switch 104 are also illustrated. The operations performed by these instructions are readily understood from a review of the drawings. The group manipulation operations illustrated in FIGS. 8(a)-8(e) comprise the presently contemplated data manipulation operations for the general purpose media processor 12. As those skilled in the art will appreciate, either a subset of these operations or additional data manipulation operations can be incorporated in other alternate embodiments of the general purpose media processor 12 without departing from the spirit and scope of the invention.

Referring again to FIG. 7, higher level mathematical operations than those performed by the multi-precision ALU 102 are performed in the general purpose media processor 12 through an extended math element 106. The extended math element 106 is coupled to the data path 108 and also comprises part of the execution unit 100. The extended math element 106 performs the complex arithmetic operations necessary for video data compression and similarly intensive mathematical operations. One presently preferred example of an extended math operation comprises a Galois field operation. Other examples of extended mathematical functions performed by the extended math element 106 include CRC generation and checking, Reed-Solomon code generation and checking, and spread-spectrum encoding and decoding. As those skilled in the art appreciate, additional mathematical operations are possible and contemplated.

According to the preferred embodiment of the integrated general purpose media processor 12, a register file 110 is provided in addition to the execution unit 100 to process media data. The register file 110 stores and transmits data

5.794.060

13

streams to and from the execution unit 100 via the data path 108. Rather than employing a complex set of specific or dedicated registers, the general purpose media processor 12 preferably includes 64 general purpose registers in the register file 110 along with one program counter (not shown). The 64 general purpose registers contained in the

14

Appendix, the contents of which are hereby incorporated herein by reference. A list of the presently preferred major operation codes for the general purpose media processor 12 appears below in Table I.

MAJOR OPERATION CODES								
MAJOR 0	32	64	96	128	160	192	224	
0	ERES	GSHUFFLEI	FMULADD16	GMULADD1	LU16LAI	SAAS64LAI	EADDIO	BFE16
1	ESHUFFLE-I4MUX	GSHUFFLE-I4MUX	FMULADD32	GMULADD2	LU16BAI	SAAS64BAI	EADDIUO	BFNUE16
2		GSELECT8	FMULADD64	GMULADD4	LU16LI	SCAS64LAI	ESETIL	BFNUGB16
3	EMDEPI	GMDEPI		GMULADD8	LU16BI	SCAS64BAI	ESETIGE	BFNUL16
4	EMUX	GMUX	FMULSUB16	GMULADD16	LU32LAI	SMAS64LAI	ESETIE	BFE32
5	EMMUX	GMMUX	FMULSUB32	GMULADD32	LU32BAI	SMAS64BAI	ESETINE	BFNUE32
6	EGFMUL64	GGFMUL8	FMULSUB64	GMULADD64	LU32LI	SMUX64LAI	ESETIUL	BFNUGE32
7	ETRANSPOSE-aMUX	GTRANSPOSE-aMUX		GEXTRACT128	LU32BI	SMUX64BAI	ESETIUGE	BFNUL32
8					L16LAI	S16LAI	ESUBIO	BFE64
9	ESWIZZLE	GSWIZZLE		GUMULADD2	L16BAI	S16BAI	ESUBIUO	BFNUE64
10		GSWIZZLECOPY		GUMULADD4	L16LI	S16LI	ESUBIL	BFNUGE64
11		GSWIZZLESWAP		GUMULADD8	L16BI	S16BI	ESUBIGE	BFNUL64
12	EDEPI	GDEPI	F.16	GUMULADD16	L32LAI	S32LAI	ESUBIE	BFE128
13	EUDEPI	GUDEPI	F.32	GUMULADD32	L32BAI	S32BAI	ESUBINE	BFNUE128
14	EWTHI	GUWTHI	F.64	GUMULADD64	L32LI	S32LI	ESUBIUL	BFNUGE128
15	EUWTHI	GUWTHI		GUEXTRACT128	L32BI	S32BI	ESUBIUGE	BFNUL128
16			GF.16	GEXTRACT1	L64LAI	S64LAI	EADDI	BANDE
17			GF.32	GEXTRACT116	L64BAI	S64BAI	EXORI	BANDNE
18			GF.64	GEXTRACT132	L64LI	S64LI	EORI	BL/BLZ
19			GF.128	GUEXTRACT164	L64BI	S64BI	EANDI	BGE/BGEZ
20			GF.16	GEXTRACT	L128LAI	S128LAI	ESUBI	BE
21			GF.32		L128BAI	S128BAI		BNE
22			GF.64	GEXTRACT	L128LI	S128LI	ENORI	BUL/BGZ
23			GF.128		L128BI	S128BI	ENANDI	BUGE/BLEZ
24				G.1	LB1	SBI		BGATEI
25				G.2	LUBI			
26				G.4				
27				G.8				
28		ECOPYI	GF.16	G.16			ECOPYI	BI
29			GF.32	G.32				BI.JNKI
30			GF.64	G.64				
31		E.MINOR	GF.128	G.128	L.MINOR	S.MINOR	E.MINOR	B.MINOR

major operation code field values

major operation code field values

register file 110 are all available to the user/programmer, and comprise a portion of the user state of the general purpose media processor 12. The general purpose registers are preferably capable of storing any form of data. Each register within the register file 110 is coupled to the data path 108 and is accessible to the execution unit 100 in the same manner. Thus, the user can employ a general purpose register according to the specific needs of a particular program or unique application. As those skilled in the art will appreciate, the register file 110 can also comprise a plurality of register files 110 configured in parallel in order to support parallel multi-threaded processing.

Instruction Set and User Programming

Control or manipulation of data processed by the general purpose media processor 12 is achieved by selected instructions programmed by the user. Those skilled in the art will appreciate that a great number of programs are possible through various sequences of instructions. Particular programs can be developed for each unique implementation of the general purpose media processor 12. A detailed discussion of such specific programs is therefore beyond the scope of this description.

One presently preferred instruction set for the general purpose media processor 12 is included in the Microfiche

As shown in Table I, the major operation codes are grouped according to the function performed by the operations. The operations are thus arranged and listed above according to the presently preferred operation code number for each instruction. As many as 255 separate operations are contemplated for the preferred embodiment of the general purpose media processor 12. As shown in Table I, however, not all of the operation codes are presently implemented. As those skilled in the art will appreciate, alternate schemes for organizing the operation codes, as well as additional operation codes for the general purpose media processor 12, are possible.

The instructions provided in the instruction set for the general purpose media processor 12 control the transfer, processing and manipulation of data streams between the register file 110 and the execution unit 100. The presently preferred width of the instruction path 112 is 32-bits wide, organized as four eight-bit bytes ("quadlets"). Those skilled in the art will appreciate, however, that the instruction path 112 can take on any width without departing from the spirit and scope of the invention. Preferably, each instruction within the instruction set is stored or organized in memory on four-byte boundaries. The presently preferred format for instructions is shown in FIG. 9(a).

As shown in FIG. 9(a), each of the presently preferred instruction formats for the general purpose media processor

5,794,060

15

12 includes a field 280 for the major operation code number shown in Table I. Based on the type of operation performed, the remaining bits can provide additional operands according to the type of addressing employed with the operation. For example, the remainder of the 32-bit instruction field can comprise an immediate operand ("imm"), or operands stored in any of the general registers ("ra," "rb," "rc," and "rd"). In addition, minor operation codes 282 can also be included among the operands of certain 32-bit instruction formats.

The presently preferred embodiment of the general purpose media processor 12 includes a limited instruction set similar to those seen in Reduced Instruction Set Computer ("RISC") systems. The preferred instruction set for the general purpose media processor 12 shown in Table I includes operations which implement load, store, synchronize, branch and gateway functions. These five groups of operations can be visually represented as two general classes of related operations. The branch and gateway operations perform related functions on media data streams and are thus visually represented as block 114 in FIG. 7. Similarly, the load, store and synchronize operations are grouped together in block 116 and perform similar operations on the media data streams. (Blocks 114 and 116 only represent the above classification of these operations and their function in the processing of media data streams, and do not indicate any specific underlying electronic connections.) A more detailed discussion of these operations, and the functionality of the general purpose media processor 12, appears in the Microfiche Appendix.

The four-byte structure of instructions for the general purpose media processor 12 is preferably independent of the byte ordering used for any data structures. Nevertheless, the gateway instructions are specifically defined as 16-byte structures containing a code address used to securely invoke a procedure at a higher privilege level. Gateways are preferably marked by protection information specified in the translation lookaside buffer 148 in the memory management unit 122. Gateways are thus preferably aligned on 16-byte boundaries in the external memory. In addition to the general purpose registers and program counter, a privilege level register is provided within the register file 110 that contains the privilege level of the currently executing instruction.

The instruction set preferably includes load and store instructions that move data between memory and the register file 110, branch instructions to compare the content of registers and transfer control, and arithmetic operations to perform computations on the contents of registers. Swap instructions provide multi-thread and multi-processor synchronization. These operations are preferably indivisible and include such instructions as add-and-swap, compare-and-swap, and multiplex-and-swap instructions. The fixed-point compare-and-branch instructions within the instruction set shown in Table I provide the necessary arithmetic tests for equality and inequality of signed and unsigned fixed-point values. The branch through gateway instruction provides a secure means to access code at a higher privileged level in a form similar to a high level language procedure call generally known in the art.

The general purpose media processor 12 also preferably supports floating-point compare-and-branch instructions. The arithmetic operations, which are supported in hardware, include floating-point addition, subtraction, multiplication, division and square root. The general purpose media processor 12 preferably supports other floating-point operations defined by the ANSI-IEEE floating-point standard through the use of software libraries. A floating point value can preferably be 16, 32, 64 or 128-bits wide. Examples of the presenting preferred floating-point data sizes are illustrated in FIG. 9(b).

16

The general purpose media processor 12 preferably supports virtual memory addressing and virtual machine operation through a memory management unit 122. Referring to FIG. 10(a), one presently preferred embodiment of the memory management unit 122 is shown. The memory management unit 122 preferably translates global virtual addresses into physical addresses by software programmable routines augmented by a hardware translation lookaside buffer ("TLB") 148. A facility for local virtual address translation 164 is also preferably provided. As those skilled in the art will appreciate, the memory management unit 122 includes a data cache 166 and a tag cache 168 that store data and tags associated with memory sections for each entry in the TLB 148.

A block diagram of one preferred embodiment of the TLB 148 is shown in FIG. 10(b). The TLB 148 receives a virtual address 230 as its input. For each entry in the TLB 148, the virtual address 230 is logically AND-ed with a mask 232. The output of each respective AND gate 234 is compared via a comparator 236 with each entry in the TLB 148. If a match is detected, an output from the comparator 236 is used to gate data 240 through a transceiver 238. As those skilled in the art will appreciate, a match indicates the entry of the corresponding physical address within the contents of the TLB 148 and no external memory or I/O access is required. The data 240 for the data cache 166 (FIG. 10(a)) is then combined with the remaining lower bits of the virtual address 230 through an exclusive-OR gate 242. The resultant combination is the physical address 244 output from the TLB 148. If a match is not detected between the logical address and the contents of the tag cache 168, the memory management unit 122 an external memory or I/O access is necessary to retrieve the relevant portion of memory and update the contents of the TLB 148 accordingly.

Using generally known memory management techniques, the memory management unit 122 ensures that instructions (and data) are properly retrieved from external memory (or other sources) over an external input/output bus 126 (see FIG. 7). As described in more detail below, a high bandwidth interface 124 is coupled to the external input/output bus 126 to communicate instructions (and media data streams) to the general purpose media processor 12. The presently preferred physical address width for the general purpose media processor 12 is eight bytes (64-bits). In addition, the memory management unit 122 preferably provides match bits (not shown) that allow large memory regions to be assigned a single TLB entry allowing for fine grain memory management of large memory sections. The memory management unit 122 also preferably includes a priority bit (not shown) that allows for preferential queuing of memory areas according to respective levels of priority. Other memory management operations generally known in the art are also performed by the memory management unit 122.

Referring again to FIG. 7, instructions received by the general purpose media processor 12 are stored in a combined instruction buffer/cache 118. The instruction buffer/cache 118 is dynamically subdivided to store the largest sequence of instructions capable of execution by the execution unit 100 without the necessity of accessing external memory. In a preferred embodiment of the invention, instruction buffer space is allocated to the smallest and most frequently executed blocks of media instructions. The instruction buffer thus helps maintain the high bandwidth capacity of the general purpose media processor 12 by sustaining the number of instructions executed per second at or near peak operation. That portion of the instruction buffer/cache 118 not used as a buffer is, therefore, available

5.794.060

17

to be used as cache memory. The instruction buffer/cache 118 is coupled to the instruction path 112 and is preferably 32 kilobytes in size.

A data buffer/cache 120 is also provided to store data transmitted and received to and from the execution unit 100 and register file 110. The data buffer/cache 120 is also dynamically subdivided in a manner similar to that of the instruction buffer/cache 118. The buffer portion of the data buffer/cache 120 is optimized to store a set size of unified media data capable of execution without the necessity of accessing external memory. In a preferred embodiment of the invention, data buffer space is allocated to the smallest and most frequently accessed working sets of media data. Like the instruction buffer, the data buffer thus maintains peak bandwidth of the general purpose media processor 12. The data buffer/cache 120 is coupled to the data path 108 and is preferably also 32 kilobytes in size.

The preferred embodiment of the general purpose media processor 12 includes a pipelined instruction pre-fetch structure. Although pipelined operation is supported, the general purpose media processor 12 also allows for non-pipelined operations to execute without any operational penalty. One preferred pipeline structure for the general purpose media processor 12 comprises a "super-string" pipeline shown in FIG. 11. A super-string pipeline is designed to fetch and execute several instructions in each clock cycle. The instructions available for the general purpose media processor 12 can be broken down into five basic steps of operation. These steps include a register-to-register address calculation, a memory load, a register-to-register data calculation, a memory store and a branch operation. According to the super-string pipeline organization of the general purpose media processor 12, one instruction from each of these five types may be issued in each clock cycle. The presently preferred ordering of these operations are as listed above where each of the five steps are assigned letters "A," "L," "E," "S" and "B" (see FIG. 11).

According to the super-string pipelining technique, each of the instructions are serially dependent, as shown in FIG. 11, and the general purpose media processor 12 has the ability to issue a string of dependent instructions in a single clock cycle. These instructions shown in FIG. 11 can take from two to five cycles of latency to execute, and a branch prediction mechanism is preferably used to keep up the pipeline filled (described below). Instructions can be encoded in unit categories such as address, load, store/sync, fixed, float and branch to allow for easy decoding. A similar scheme is employed to pre-fetch data for the general purpose media processor 12.

As those skilled in the art will appreciate, the super-string pipeline can be implemented in a multi-threaded environment. In such an implementation, the number of threads is preferably relatively prime with respect to functional unit rates so that functional units can be scheduled in a non-interfering fashion between each thread.

In another more preferred embodiment, a "super-spring" pipelining scheme is employed with the general purpose media processor 12. The super-spring pipeline technique breaks the super-string pipeline shown in FIG. 11 into two sections that are coupled via a memory buffer (not shown). A visual representation of the super-spring pipeline technique is shown in FIG. 12. The front of the pipeline 204, in which address calculation (A), memory load (L), and branch (B) operations are handled, is decoupled from the back of the pipeline 206, in which data calculation (E) and memory store (S) operations are handled. The decoupling is accomplished through the memory buffer (not shown), which is

18

preferably organized in a first-in-first-out ("FIFO") fast/dense structure. (The memory buffer is functionally represented as a spring in FIG. 12.)

As indicated in Table I above, the general purpose media processor 12 does not include delayed branch instructions, and so relies upon branch or fetch prediction techniques to keep the pipeline full in program flows around unconditional and conditional branch instructions. Many such techniques are generally known in the art. Examples of some presently preferred techniques include the use of group compare and set, and multiplex operations to eliminate unpredictable branches; the use of short forward branches, which cause pipeline neutralization; and where branch and link predicts the return address in a one or more entry stack. In addition, the specialized gateway instructions included in the general purpose media processor 12 allow for branches to and from protected virtual memory space. The gateway instructions, therefore, allow an efficient means to transfer between various levels of privilege.

As described above, two basic forms of media data are processed by the general purpose media processor 12, as shown in FIG. 7. These data streams generally comprise Nyquist sampled I/O 128, and standard memory and I/O 130. As shown in FIG. 7, audio 132, video 134, radio 136, network 138, tape 140 and disc 142 data streams comprise some examples of digitally sampled I/O 128. As those skilled in the art will appreciate, other forms of digitally sampled I/O are contemplated for processing by the general purpose media processor 12 without departing from the spirit and scope of the invention. Standard memory and I/O 130 comprises data received and transmitted to and from general digital peripheral devices used in the design of most computer systems. As shown in FIG. 7, some examples of such devices include dynamic random access memory ("DRAM") 146, or any data received over the PCI bus 144 generally known in the art. Other forms of standard memory and I/O sources are also contemplated. The various fixed-point data sizes preferred for the general purpose media processor 12 are illustrated in FIG. 9(c).

External Interface

As mentioned above, the general purpose media processor 12 includes a high bandwidth interface 124 to communicate with external memory and input/output sources. As part of the high bandwidth interface 124, the general purpose media processor 12 integrates several fast communication channels 156 (FIG. 13) to communicate externally. These fast communication channels 156 preferably couple to external caches 150, which serve as a buffer to memory interfaces 152 coupled to standard memory 154. The caches 150 preferably comprise synchronous static random access memory ("SRAM"), each of which are sixty-four kilobytes in size; and the standard memories 154 comprise DRAM's. The memory interfaces 152 transmit data between the caches 150 and the standard memories 154. The standard memories 154 together form the main external memory for the general purpose media processor 12. The cache 150, memory interface 152, standard memory 154 and input/output channel 156 therefore make up a single external memory unit 158 for the general purpose media processor 12.

According to the presently preferred embodiment of the invention, the memory interface protocol embeds read and write operations to a single memory space into packets containing command, address, data and acknowledgment information. The packets preferably include check codes that will detect single-bit transmission errors and some

5.794.060

19

multiple-bit errors. As many as eight operations may be in progress at a time in each external memory unit 158. As shown in FIG. 13, up to four external memory units 158 may be cascaded together to expand the memory available to the general purpose media processor 12, and to improve the bandwidth of the external memory. Through such cascaded memory units 158, the memory interface 152 provides for the direct connection of multiple banks of standard memory 154 to maintain operation of the general purpose media processor 12 at sustained peak bandwidths.

According to one embodiment shown in FIG. 13, up to four standard memory devices 154 can be coupled to each memory interface 152. Each standard memory 154 thus includes as many as four banks of DRAM, each of which is preferably sixteen bits wide. The standard memories 154 are connected in parallel to the memory interface 152 forming a 72-bit wide data bus 160, where 64 bits are preferably provided for data transfer and eight bits are provided for error correction. In addition to the data bus 160, an address/control bus 162 is coupled between the memory interface 152 and each standard memory 154. The address/control bus 162 preferably comprises at least twelve address lines (4 kilobits \times 16 memory size) and four control lines as shown in FIG. 13. An alternate manner for coupling the DRAM's to the memory interface 152 is illustrated in FIG. 14. As shown in FIG. 14, two banks of four DRAM single in-line memory modules are coupled in parallel to the memory interface 152. The memory interface 152 also supports interleaving to enhance bandwidth, and page mode accesses to improve latency for localized addressing.

Using standard DRAM components, the external memory units 158 achieve bandwidths of approximately two gigabits/second with the standard memories 154. When four such external memory units 158 are coupled via the communication channel 156, therefore, the total bandwidth of the external main memory system increases to one gigabyte/second. As discussed further below, in implementations with two or eight communication channels 156, the aggregate bandwidth increases to two and eight gigabytes/second, respectively.

A more detailed depiction of the communication channel 156 circuitry appears in FIG. 15. According to the preferred embodiment of the invention, each communication channel 156 comprises two unidirectional, byte-wide, differential, packet-oriented data channels 156a, 156b (see FIG. 13). As explained above, where memory units 158 are cascaded together in series, the output of one memory unit 158 is connected to the input of another memory unit 158. The two unidirectional channels are thus connected through the memory units 158 forming a loop structure and make up a single bi-directional memory interface channel.

Referring to FIG. 15, each communication channel 156 is preferably eight bits wide, and each bit is transmitted differentially. For example, output transceiver 170 for bit D_{0out} transmits both D_0 and $\overline{D_0}$ signals over the communication channel 156. Additional transceivers are similarly provided for the remaining bits in the channel 156. (The transceiver 176 for bit D_{7out} and associated differential lines 178, 180 are shown in FIG. 15.) A CLK_{out} transceiver 182 is also provided to generate differential clock outputs 184, 186 over the channel 156. To complete the link between memory units 158, input transceivers 188–192 are provided in each memory unit 158 for each of the differential bits and clock signals transmitted over the communication channel 156. These input signals 172, 174, 178, 180, 184, 186 are preferably transmitted through input buffers 194–198 to other parts of the memory unit 158 (described above).

20

Each memory unit 158 also includes a skew calibrator 200 and phase locked loop ("PLL") 202. The skew calibrator 200 is used to control skew in signals output to the communication channel 156. Preferably, digital skew fields are employed, which include set numbers of delay stages to be inserted in the output path of the communication channel 156. Setting these fields, and the corresponding analog skew fields, permits a fine level of control over the relative skew between output channel signals.

The PLL 202 recovers the clock signal on either side of the communication channel 156 and is thus provided to remove clock jitter. The clock signals 184, 186 preferably comprise a single phase, constant rate clock signal. The clock signals 184, 186 thus contain alternating zero and one values transmitted with the same timing as the data signals 172, 174, 178, 180. The clock signal frequency is, therefore, one-half the byte data rate. The communication channel 156 preferably operates at constant frequency and contains no auxiliary control, handshaking or flow control information.

Each external memory unit 158 preferably defines two functional regions: a memory region, implemented by the cache 150 backed by standard memory 154 (see FIG. 13), and a configuration region, implemented by registers (not shown). Both regions are accessed by separate interfaces; the communication channel 156 is used to access the memory region, and a serial interface (described below) is used to access the configuration region. In the memory region, the caches 150 are preferably write-back (write-in) single-set (direct-map) caches for data originally contained in standard memory 154. All accesses to memory space should maintain consistency between the contents of the cache 150 and the contents of the standard memory 154. The configuration region registers provide the mechanism to detect and adjust skew in the communication channel 156. Software is preferably employed to adaptively adjust the skew in the channel 156 through digital skew fields, as explained above. The serial interface thus is used to configure the external memory units 158, set diagnostic modes and read diagnostic information, and to enable the use of a high-speed tester (not shown).

One presently preferred embodiment of the invention employs two byte-wide packet communication channels 156 (FIG. 16(a)). In order to further increase the bandwidth of the general purpose media processor 12, up to sixteen byte-wide packet communication channels 156 can be employed. Referring to FIG. 16(b), twelve communication channels, comprising eight memory channels 210, a ninth channel for parallel processing 212 (described below), and three input/output ("I/O") channels 214, are shown. Each of the communication channels 210–214 preferably employs the cascade configuration of four channel interface devices 216. (Each channel interface device 216 coupled to the memory channels 210 corresponds to the external memory unit 158 shown in FIG. 13.) Through each of the twelve communication channels shown in FIG. 16(b), the general purpose media processor 12 can request or issue read or write transactions. When not interleaved, the twelve channels provide a single contiguous memory space for each channel interface device 216.

Alternatively, memory accesses may be interleaved in order to provide for continuous access to the external memory system at the maximum bandwidth for the DRAM memories. In an interleaved configuration, at any point in time some memory devices will be engaged in row pre-charge, while others may be driving or receiving data, or receiving row or column addresses. The memory interface 152 (FIG. 13) thus preferably maps between a contiguous

5,794,060

21

address space and each of the separate address spaces made available within each external memory unit 158. For maximum performance, therefore, the memory interface is interleaved so that references to adjacent addresses are handled by different memory devices. Moreover, in the preferred embodiment, additional memory operations may be requested before the corresponding DRAM bank is available. In an interleaved approach, these operations are placed in a queue until they can be processed. According to the preferred embodiment, memory writes have lower priority than memory reads, unless an attempt is made to read an address that is queued for a write operation. As those skilled in the art will appreciate, the depth of the memory write queue is dictated by the specific implementation.

Although up to four external memory units 158 are preferably cascaded to form effectively larger memories, some amount of latency may be introduced by the cascade. Packets of data transmitted over the communication channel 156 are uniquely addressed to a particular channel interface device 216. A packet received at a particular device, which specifies another module address, is automatically passed to the correct channel interface device 216. Unless the module address matches a particular device 216, that packet simply passes from the input to the output of the interface device 216. This mechanism divides the serial interconnection of interface devices 216 into strings, which function as a single larger memory or peripheral, but with possibly longer response latency.

In addition to the memory channels 210, the general purpose media processor 12 provides several communication channels 214 for communication with external input/output devices. Referring to FIG. 16(b), three input/output channels 214 having SRAM buffered memory (see FIG. 13) provide an interface to external standard I/O devices (not shown). Like the eight memory channels 210, the three I/O channels 214 are byte-wide input/output channels intended to operate at rates of at least one gigahertz. The three I/O channels 214 also operate as a packet communication link to synchronous SRAM memory 208 within the channel interface device 216. A controller 226 within the channel interface device 216 completes the interface to the I/O devices.

The three I/O channels 214 preferably function in like manner to the memory channels 210 described above. The interface protocol for the three I/O channels 214 divides read and write operations to a single memory space into packets containing command, address, data and acknowledgment information. The packets also include a check code that will detect single-bit transmission errors and some multiple-bit errors. According to the preferred embodiment of the invention, as many as eight operations may progress in each interface device 216 at a time. As shown in FIG. 16(b), up to four channel interface devices 216 can be cascaded together to expand the bandwidth in the three I/O channels 214. A bit-serial interface (not shown) is also provided to each of the channel interface devices 216 to allow access to configuration, diagnostic and tester information at standard TTL signal levels at a more moderate data rate. (A more detailed description of the serial interface is provided below).

Like the memory channels 210, each I/O channel 214 includes nine signals—one clock signal and eight data signals. Differential voltage levels are preferably employed for each signal. Each channel interface device 216 is preferably terminated in a nominal 50 ohm impedance to ground. This impedance applies for both inputs and outputs to the communication channel 156. A programmable termination impedance is preferred.

22

Interface Communication

According to one presently preferred embodiment of the invention, the channel interface devices 216 can operate as either master devices or slave devices. A master device is capable of generating a request on the communication channel 156 and receiving responses from the communication channel 156. Slave devices are capable of receiving requests and generating responses over the communication channel 156. A master device is preferably capable of generating a constant frequency clock signal and accepting signals at the same clock frequency over the communication channel 156. A slave device, therefore, should operate at the same clock rate as the communication channel 156, and generate no more than a specified amount of variation in output clock phase relative to input clock phase. The master device, however, can accept an arbitrary input clock phase and tolerates a specified amount of variation in clock phase over operating conditions.

Packets of information sent over the communication channel 156 preferably contain control commands, such as read or write operations, along with addresses and associated data. Other commands are provided to indicate error conditions and responses to the above commands. When the communication channel 156 is idle, such as during initialization and between transmitted packets, an idle packet, consisting of an all-zero byte and an all-one byte is transmitted through the communication channel 156. Each non-idle packet consists of two bytes or a multiple of two bytes, and begins with a byte having a value other than all zeros. All packets transmitted over the communication channel 156 also begin during a clock period in which the clock signal is zero, and all packets preferably end during a clock period in which the clock signal is one. A depiction of the preferred packet protocol format for transmission over the communication channel 156 appears in FIG. 17.

The general form of each packet is an array of bytes preferably without a specific byte ordering. The first byte contains a module address 250 ("ma") in the high order two bits; a packet identifier, usually a command 252 ("com"), in the next three bit positions; and a link identification number 254 ("lid") in the last three bit positions. The interpretation of the remaining bytes of a packet depend upon the contents of the packet identifier. The length of each packet is preferably implied by the command specified in the initial byte of the packet. A check byte is provided and computed as odd bit-wise parity with a leftward circular rotation after accumulating each byte. This technique provides detection of all single-bit and some multiple-bit errors, but no correction is provided.

The modular address 250 field of each packet is preferably a two-bit field and allows for as many as four slave devices to be operated from a single communication channel 156. Module address values can be assigned in one of two fashions: either dynamically assigned through a configuration register (not shown), or assigned via static/geometric configuration pins. Dynamic assignment through a configuration register is the presently preferred method for assigning module address values.

The link identification number 254 field is preferably 3-bits wide and provides the opportunity for master devices to initiate as many as eight independent operations at any one time to each slave device. Each outstanding operation requires a distinct link identification number, but no ordering of operations should be implied by the value of the link identification field. Thus, there is preferably no requirement for link identification values 254 to be sequentially assigned either in requests or responses.

5,794,060

23

The receipt of packets over the communication channel 156 that do not conform to the channel protocol preferably generates an error condition. As those skilled in the art will appreciate, the level or degrees to which a specific implementation detects errors is defined by the user. In one presently preferred embodiment of the invention, all errors are detected, and the following protocol is employed for handling errors. For each error detected, the channel interface device 216 causes a response explicitly indicating the error condition. Channel interface devices 216 reporting an invalid packet will then suppress the receipt of additional packets until the error is cleared. The transmitted packet is otherwise ignored. However, even though the erroneous packet is ignored, the channel interface devices 216 preferably continue to process valid packets that have already been received and generate responses thereto. An identification of the presently preferred commands 252 to be used over the communication channel 156 are listed in FIG. 17.

In the master/slave preferred embodiment, the channel interface devices 216 forward packets that are intended for other devices connected to the communication channel 156, as described above. In slave devices, forwarding is performed based on the module address 250 field of the packet. Packets which contain a module address 250 other than that of the current device are forwarded on to the next device. All non-idle packets are thus forwarded including error packets. In master devices, forwarding is performed based on the link identifier number 254 of the packet. Packets that contain link identifier numbers 254 not generated by the specific channel interface device 216 are forwarded. In order to reduce transmission latency, a packet buffer may be provided. As those skilled in the art appreciate, the suitable size for the packet buffer depends on the amount of latency tolerable in a particular implementation.

A variety of master/slave ring configurations are possible using the high bandwidth interface 124 of the invention. Five ring configurations are currently preferred: single-master, dual-master, multiple-master, single-slave and multiple-master/multiple-slave. The simplest ring configuration contains a single non-forwarding master device and a single non-forwarding slave device. No forwarding is required for either device in this configuration as packets are sent directly to the recipient. A single-master ring, however, may contain a cascade of up to four slave devices (see FIGS. 13, 16). In the single-master ring configuration, each slave device is configured to a distinct module address, and each slave device forwards packets that contain module address fields unequal to their own. As discussed above, a single-master ring provides a larger memory or I/O capacity than a master-slave pair, but also introduces a potentially longer response latency. In the single-master ring, each slave device may have as many as eight transactions outstanding at any time, as described above.

The remaining combinations share many of the above basic attributes. In a dual-master pair, each master device may initiate read and write operations addressed to the other, and each may have up to eight such transactions outstanding. No forwarding is required for either device because packets are sent directly to the recipient. A multiple-master ring may contain multiple master devices and a single slave device. In this configuration, the slave device need not forward packets as all input packets are designated for the single slave device. A multiple-master ring may contain multiple master devices and as many as four slave devices. Each slave device may have up to eight transactions outstanding, and each master device may use some of those transactions. In a preferred embodiment, a master also has the capability to

24

detect a time-out condition or when a response to a request packet is not received. Further aspects of inter-processor communications and configurations are discussed below in connection with FIG. 18.

Serial Bus

In one preferred embodiment of the invention, the general purpose media processor 12 includes a serial bus (not shown). The serial bus is designed to provide bootstrap resources, configuration, and diagnostic support to the general purpose media processor 12. The serial bus preferably employs two signals, both at TTL levels, for direct communication among many devices. In the preferred embodiment, the first signal is a continuously running clock, and the second signal is an open-collector bi-directional data signal. Four additional signals provide geographic addresses for each device coupled to the serial bus. A gateway protocol, and optional configurable addressing, each provide a means to extend the serial bus to other buses and devices. Although the serial bus is designed for implementation in a system having a general purpose media processor 12, as those skilled in the art will appreciate, the serial bus is applicable to other systems as well.

Because the serial bus is preferably used for the initial bootstrap program load of the general purpose media processor 12, the bootstrap ROM is coupled to the serial bus. As a result, the serial bus needs to be operational for the first instruction fetch. The serial bus protocol is therefore devised so that no transactions are required for initial bus configuration or bus address assignment.

According to the preferred embodiment, the clock signal comprises a continuously running clock signal at a minimum of 20 megahertz. The amount of skew, if any, in the clock signal between any two serial bus devices should be limited to be less than the skew on the data signal. Preferably, the serial data signal is a non-inverted open collector bi-directional data signal. TTL levels are preferred for communication on the serial bus, and several termination networks may be employed for the serial data signal. A simple preferred termination network employs a resistive pull-up of 220 ohms to 3.3 volts above V_{cc} . An alternate embodiment employs a more complex termination network such as a termination network including diodes or the "Forced Perfect Termination" network proposed for the SCSI-2 standard, which may be advantageous for larger configurations.

The geographic addressing employed in the serial bus is provided to insure that each device is addressable with a number that is unique among all devices on the bus and which also preferably reflects the physical location of the device. Thus, the address of each device remains the same each time the system is operated. In one preferred embodiment, the geographic address is composed of four bits, thus allowing for up to 16 devices. In order to extend the geographic addressing to more than 16 devices, additional signals may be employed such as a buffered copy of the clock signal or an inverted copy of the clock signal (or both).

The serial bus preferably incorporates both a bit level and packet protocol. The bit level protocol allows any device to transmit one bit of information on the bus, which is received by all devices on the bus at the same time. Each transmitted bit begins at the rising edge of the clock signal and ends at the next rising edge. The transmitted bit value is sampled at the next rising edge of the clock signal. According to one preferred embodiment where the serial data signal is an open

5,794,060

25

collector signal, the transmission of a zero bit value on the bus is achieved by driving the serial data signal to a logical low value. In this embodiment, the transmission of a one bit value is achieved by releasing the serial data signal to obtain a logical high value. If more than one device attempts to transmit a value on the same clock, the resulting value is a zero if any device transmits a zero value, and one if all devices transmit a one value. This provides a "wired-AND" collision mechanism, as those skilled in the art will appreciate. If two or more devices transmit the same value on the same clock cycle, however, no device can detect the occurrence of a collision. In such cases, the transaction, which may occur frequently in some implementations, preferably proceeds as described below.

The packet protocol employed with the serial bus uses the bit level protocol to transmit information in units of eight bits or multiples of eight bits. Each packet transmission preferably begins with a start bit in which the serial data signal has a zero (driven) value. After transmitting the eight data bits, a parity bit is transmitted. The transmission continues with additional data. A single one (released) bit is transmitted immediately following the least significant bit of each byte signaling the end of the byte.

On the cycle following the transmission of the parity bit, any device may demand a delay of two cycles to process the data received. The two cycle delay is initiated by driving the serial data signal (to a zero value) and releasing the serial data signal on the next cycle. Before releasing the serial data signal, however, it is preferable to insure that the signal is not being driven by any other device. Further delays are available by repeating this pattern.

In order to avoid collisions, a device is not permitted to start a transmission over the serial bus unless there are no currently executing transactions. To resolve collisions that may occur if two devices begin transmission on the same cycle, each transmitting device should preferably monitor the bus during the transmission of one (released) bits. If any of the bits of the byte are received as zero when transmitting a one, the device has lost arbitration and must cease transmission of any additional bits of the current byte or transaction.

According to the preferred embodiment of the invention, a serial bus transaction consists of the transmission of a series of packets. The transaction begins with a transmission by the transaction initiator, which specifies the target network, device, length, type and payload of the transaction request. The transaction terminates with a packet having a type field in a specified range. As a result, all devices connected to the serial bus should monitor the serial data signal to determine when transactions begin and end. A serial bus network may have multiple simultaneous transactions occurring, however, so long as the target and initiator network addresses are all disjoint.

Parallel Processing

In one preferred embodiment of the invention, two or more general purpose media processors 12 can be linked together to achieve a multiple processor system. According to this embodiment, general purpose media processors 12 are linked together using their high bandwidth interface channels 124, either directly or through external switching components (not shown). The dual-master pair configuration described above can thus be extended for use in multiple-master ring configurations. Preferably, internal daemons provide for the generation of memory references to remote processors, accesses to local physical memory space, and the transport of remote references to other remote processors. In a multi-processor environment, all general purpose media processors 12 run off of a common clock frequency, as

26

required by the communication channels 156 that connect between processors.

Referring to FIG. 18, each general purpose media processor 12 preferably includes at least a pair of inter-processor links 218 (see also FIG. 16(b)). In one configuration, both pairs of inter-processor links 218 can be connected between the two processors 12 to further enhance bandwidth. As shown in FIG. 18(a) several processors 12 may be interconnected in a linear network employing the transponder daemons in each processor. In an alternate embodiment shown in FIG. 18(b), the inter-processor links 218 may be used to join the general purpose media processors 12 in a ring configuration. Alternatively still, general purpose media processors 12 may be interconnected into a two-dimensional network of processors of arbitrary size, as shown in FIG. 18(c). Sixteen processors are connected in FIG. 18(c) by connecting four ring networks. In yet another alternate embodiment, by connecting the inter-processor links 218 to external switching devices (not shown), multi-processors with a large number of processors can be constructed with an arbitrary interconnection topology.

The requester, responder and transponder daemons preferably handle all inter-processor operations. When one general purpose media processor 12 attempts a load or store to a physical address of a remote processor, the requester daemon autonomously attempts to satisfy the remote memory reference by communicating with the external device. The external device may comprise another processor 12 or a switching device (not shown) that eventually reaches another processor 12. Preferably, two requester daemons are provided each processor 12, which act concurrently on two different byte channels and/or module addresses. The responder daemon accepts writes from a specified channel and module address, which enables an external device to generate transaction requests in local memory or to generate processor events. The responder daemon also generates link level writes to the same external device that communicated responses for the received transaction request. Two such responder daemons are preferably provided; each of which operate concurrently to two different byte channels and/or module addresses.

The transponder daemon accepts writes from a specified channel and module address, which enable an external device to cause a requester daemon to generate a request on another channel and module address. Preferably, two such transponder daemons are provided, each of which act concurrently (back-to-back) between two different byte channel and/or module addresses. As those skilled in the art will appreciate, the requester, responder and transponder daemons must act cooperatively to avoid deadlock that may arise due to an imbalance of requests in the system. Deadlocks prevent responses from being routed to their destinations, which may defeat the benefits of a multi-processor distributed system.

According to one presently preferred embodiment of the invention, the general purpose media processor 12 can be implemented as one or more integrated circuit chips. Referring to FIG. 19, the presently preferred embodiment of the general purpose media processor 12 consists of a four-chip set. In the four-chip set, a general purpose media processor 12 is manufactured as a stand alone integrated circuit. The stand alone integrated circuit includes a memory management unit 122, instruction and data cache/buffers 118, 120, and an execution unit 100. A plurality of signal input/output pads 260 are provided around the circumference of the integrated circuit to communicate signals to and from the general purpose media processor 12 in a manner generally known in the art.

The second and third chips of the four-chip set comprise in an external memory element 158 and a channel interface

5,794,060

27

device 216. The external memory element 158 includes an interface to the communication channel 156, a cache 150 and a memory interface 152. The channel interface device 216 also includes an interface to the communication channel 156, as well as buffer memory 262, and input/output interfaces 264. Both the external memory element 158 and the channel interface device 216 include a plurality of input/output signal pads 260 to communicate signals to and from these devices in a generally known manner.

The fourth integrated circuit chip comprises a switch 226, which allows for installation of the general purpose media processor 12 in the heterogeneous network 38. In addition to the plurality of input/output pads 260, the switch 226 includes an interface to the communication channel 156. The switch 226 also preferably includes a buffer 262, a router 266, and a switch interface 268.

As those skilled in the art will appreciate, many implementations for the general purpose media processor 12 are possible in addition to the four-chip implementation described above. Rather than an integrated approach, the general purpose media processor can be implemented in a discrete manner. Alternatively, the general purpose media processor 12 can be implemented in a single integrated circuit, or in an implementation with fewer than four integrated circuit chips. Other combinations and permutations of these implementations are contemplated.

There has been described a system for processing streams of media data at substantially peak rates to allow for real time communication over a large heterogeneous network. The system includes a media processor at its core that is capable of processing such media data streams. The heterogeneous network consists of, for example, the fiber optic/coaxial cable/twisted wire network in place throughout the U.S. To provide for such communication of media data, a media processor according to the invention is disposed at various locations throughout the heterogeneous network. The media processor would thus function both in a server capacity and at an end user site within the network. Examples of such end user sites include televisions, set-top converter boxes, facsimile machines, wireless and cellular telephones, as well as large and small business and industrial applications.

To achieve such high rates of data throughput, the media processor includes an execution unit, high bandwidth interface, memory management unit, and pipelined instruction and data paths. The high bandwidth interface includes a mechanism for transmitting media data streams to and from the media processor at rates at or above the gigahertz frequency range. The media data stream can consist of transmission, presentation and storage type data transmitted alone or in a unified manner. Examples of such data types include audio, video, radio, network and digital communications. According to the invention, the media processor is dynamically partitionable to process any combination or permutation of these data types in any size.

A programmable, general purpose media processor system presents significant advantages over current multimedia communications. Rather than rigid, costly and inefficient specialized processors, the media processor provides a general purpose instruction set to ease programmability in a single device that is capable of performing all of the operations of the specialized processor combination. Providing a uniform instruction set for all media related operations eliminates the need for a programmer to learn several different instruction sets, each for a different specialized processor. The complexity of programming the specialized processors to work together and communicate with one another is also greatly reduced. The unified instruction set is also more efficient. Highly specialized general calculation

28

instructions that are tailored to general or special types of calculations rather than enhancing communication are eliminated.

Moreover, the media processor system can be easily reprogrammed simply by transmitting or downloading new software over the network. In the specialized processor approach, new programming usually requires the delivery and installation of new hardware. Reprogramming the media processor can be done electronically, which of course is quicker and less costly than the replacement of hardware.

It is to be understood that a wide range of changes and modifications to the embodiments described above will be apparent to those skilled in the art and are contemplated. It is therefore intended that the foregoing detailed description be regarded as illustrative rather than limiting, and that it be understood that it is the following claims, including all equivalents, that are intended to define the spirit and scope of this invention.

We claim:

1. A method for processing unified media data streams, comprising the steps of:

receiving a plurality of unified media data streams transmitted over a data path, including presentation, transmission and storage information;

dynamically partitioning the unified media data streams based on an elemental symbol width, said elemental symbol width being equal to or narrower than the data path; and

processing the unified media data streams at substantially peak operation; and

wherein the step of processing the unified media data streams further comprises the steps of:

storing the unified media data streams in a general register file;

performing multi-precision operations on the stored unified media data streams based on programmed instructions, the multi-precision arithmetic operations including boolean, integer and floating point mathematical operations;

manipulating component fields of the unified media data streams based on programmed instructions that implement copying, shifting and re-sizing operations; and performing multi-precision mathematical operations on the stored unified media data streams based on programmed instructions, the mathematical operations including finite group, finite ring and table look-up operations.

2. The method defined in claim 1, further comprising the steps of:

pre-fetching instructions and data to fill instruction and data pipelines;

performing memory management operations to retrieve instructions and data from external memory;

storing instructions and data in instruction and data cache/buffers; and

dynamically allocating buffer storage in the instruction and data cache/buffers to ensure real-time execution.

3. The method defined in claim 1, further comprising the step of providing a set of instructions to process the unified media data streams, the set of instructions including load, store, synchronization, branch and gateway instructions.

4. The method defined in claim 3, further comprising the step of programming a sequence of at least one instruction from the set of instructions.

* * * * *

EXHIBIT 7



US005794061A

United States Patent [19]

Hansen et al.

[11] Patent Number: 5,794,061
 [45] Date of Patent: Aug. 11, 1998

[54] GENERAL PURPOSE, MULTIPLE PRECISION PARALLEL OPERATION, PROGRAMMABLE MEDIA PROCESSOR

[75] Inventors: Craig Hansen, Los Altos; John Moussouris, Palo Alto, both of Calif.

[73] Assignee: Microunity Systems Engineering, Inc., Sunnyvale, Calif.

[21] Appl. No.: 754,829

[22] Filed: Nov. 22, 1996

Related U.S. Application Data

[62] Division of Ser. No. 516,036, Aug. 16, 1995, Pat. No. 5,742,840.

[51] Int. Cl.⁶ G06F 9/00

[52] U.S. Cl. 395/800.01

[58] Field of Search 395/800.01, 670, 395/376, 280; 364/131-134, 736, 741, 745, 748, 754, 761, 768, 736.01, 745.01, 748.01, 754.01

References Cited

U.S. PATENT DOCUMENTS

4,893,267	1/1990	Alsup et al.	364/745
4,975,868	12/1990	Freerksen	364/478
5,201,056	4/1993	Daniel et al.	395/800
5,268,855	12/1993	Mason et al.	364/746
5,426,600	6/1995	Nakagawa et al.	364/764

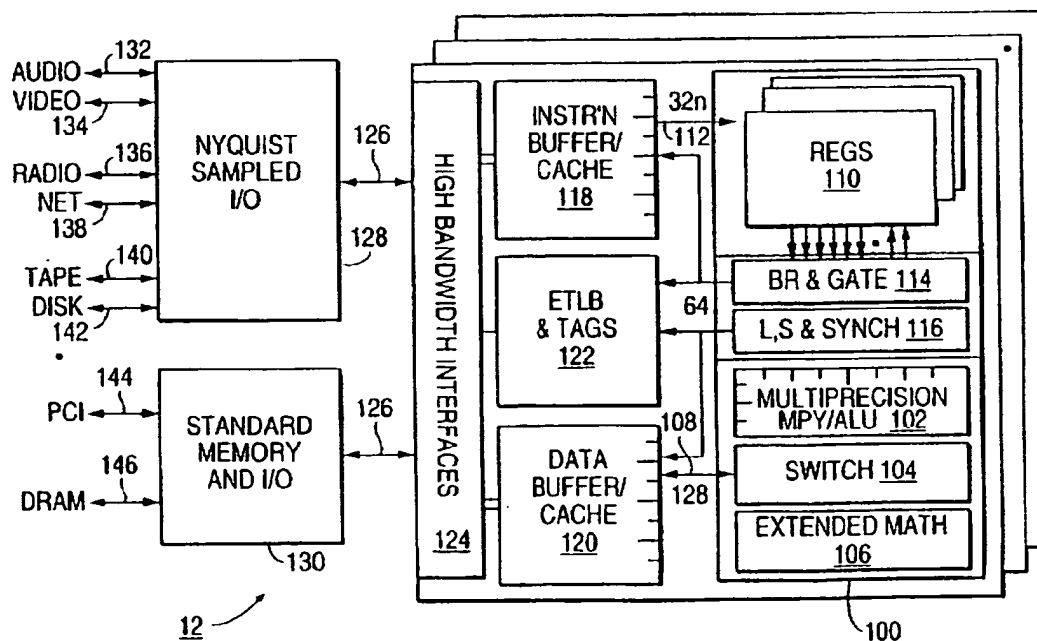
Primary Examiner—Alpesh M. Shah
 Attorney, Agent, or Firm—McDermott, Will & Emery

[57] ABSTRACT

A general purpose, programmable media processor for processing and transmitting a media data stream of audio, video, radio, graphics, encryption, authentication, and networking information in real-time. The media processor incorporates an execution unit that maintains substantially peak data throughout of media data streams. The execution unit includes a dynamically partitionable multi-precision arithmetic unit, programmable switch and programmable extended mathematical element. A high bandwidth external interface supplies media data streams at substantially peak rates to a general purpose register file and the multi-precision execution unit. A memory management unit, and instruction and data cache/buffers are also provided. High bandwidth memory controllers are linked in series to provide a memory channel to the general purpose, programmable media processor. The general purpose, programmable media processor is disposed in a network fabric consisting of fiber optic cable, coaxial cable and twisted pair wires to transmit, process and receive single or unified media data streams. Parallel general purpose media processors are disposed throughout the network in a distributed virtual manner to allow for multi-processor operations and sharing of resources through the network. A method for receiving, processing and transmitting media data streams over the communications fabric is also provided.

30 Claims, 25 Drawing Sheets

Microfiche Appendix Included
 (4 Microfiche, 387 Pages)



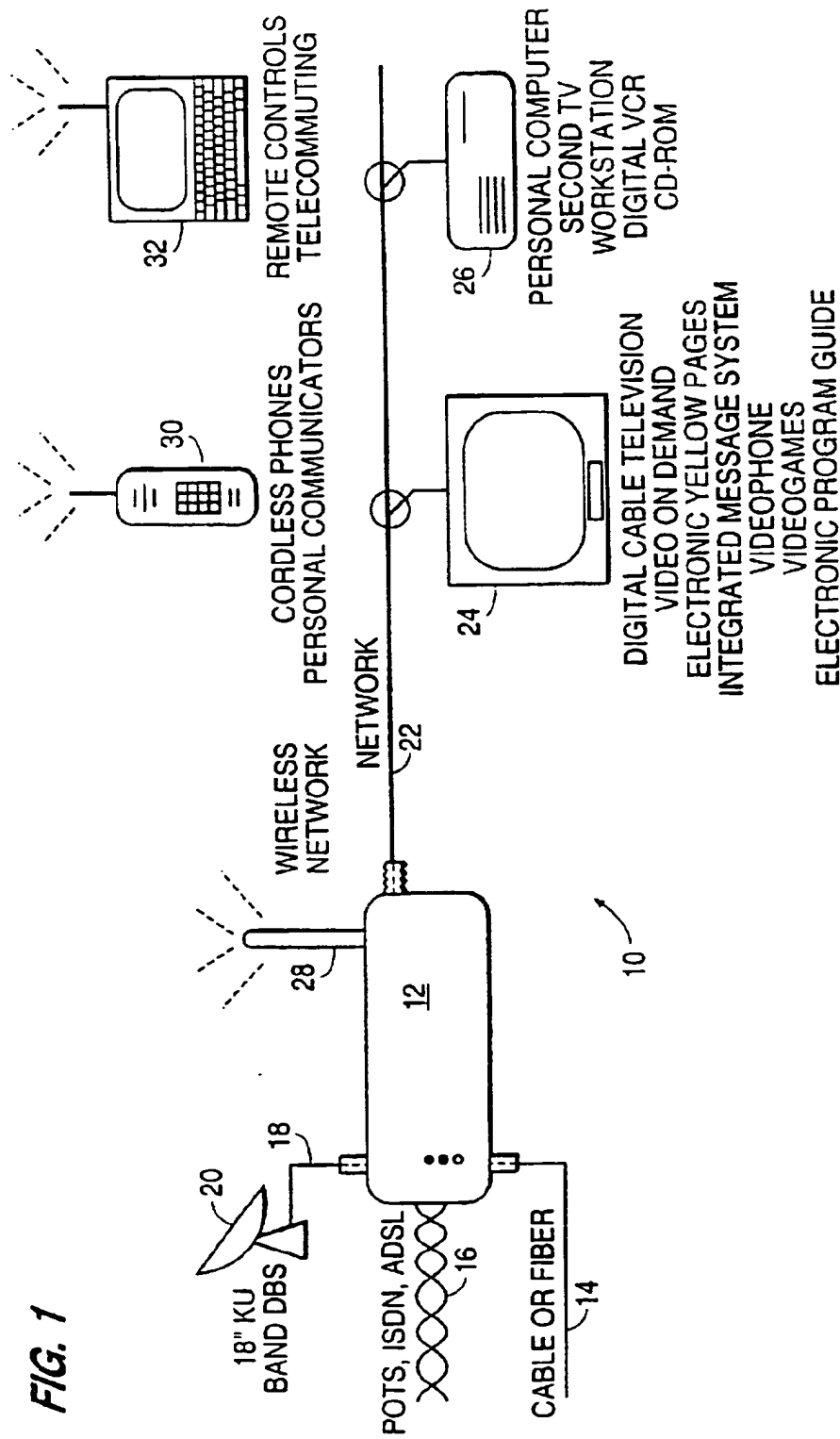
U.S. Patent

Aug. 11, 1998

Sheet 1 of 25

5,794,061

FIG. 1



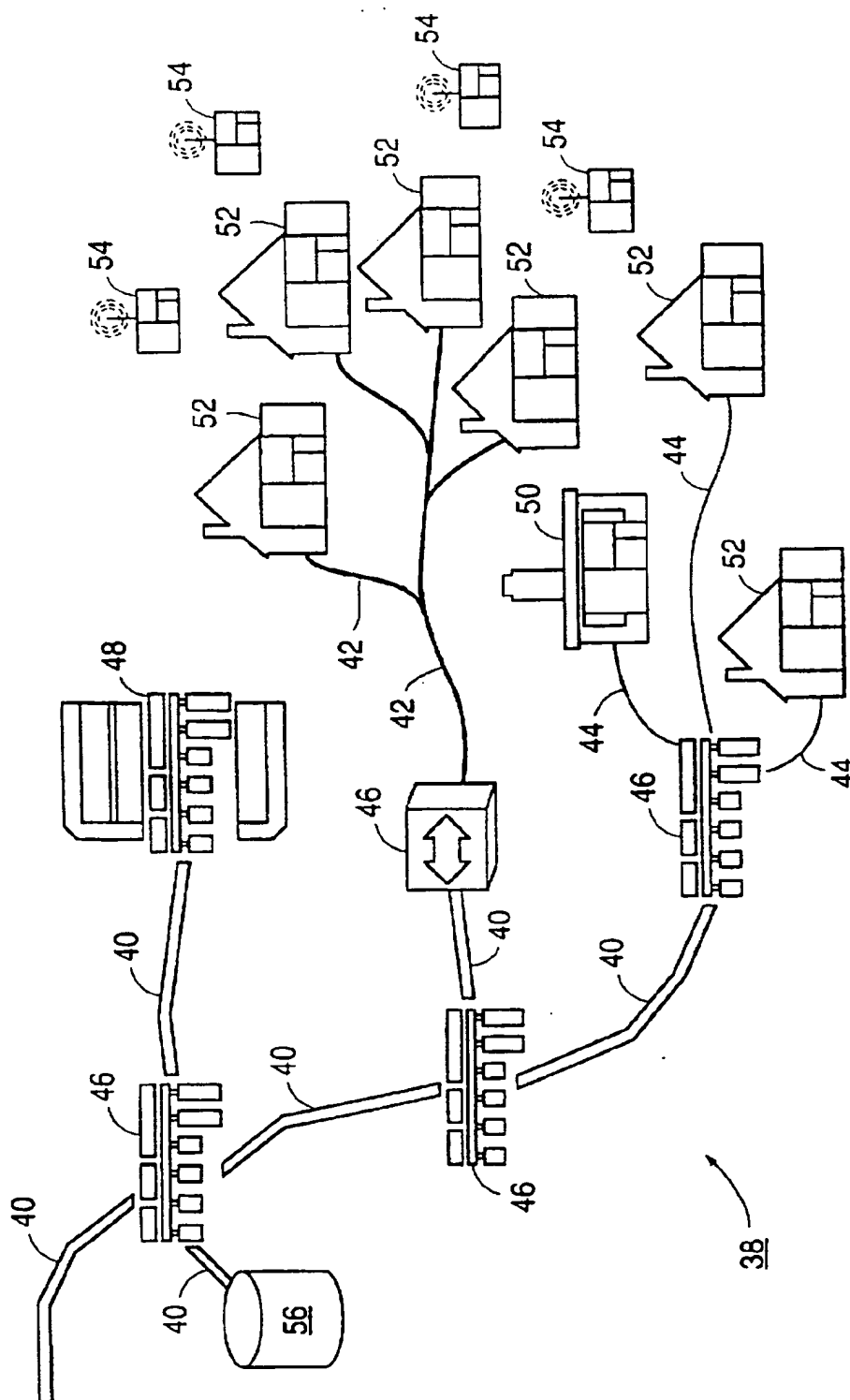
U.S. Patent

Aug. 11, 1998

Sheet 2 of 25

5,794,061

FIG. 2

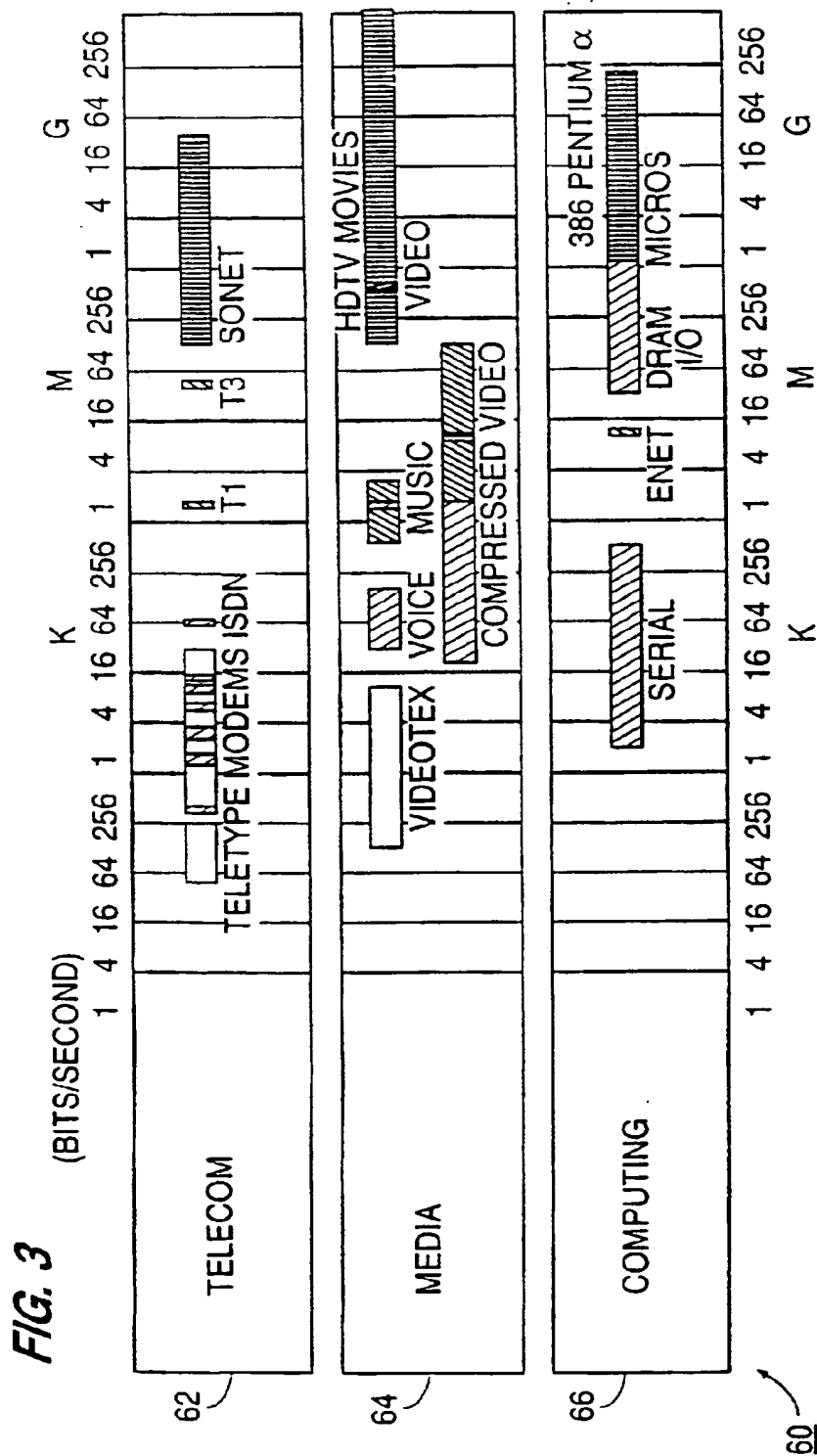


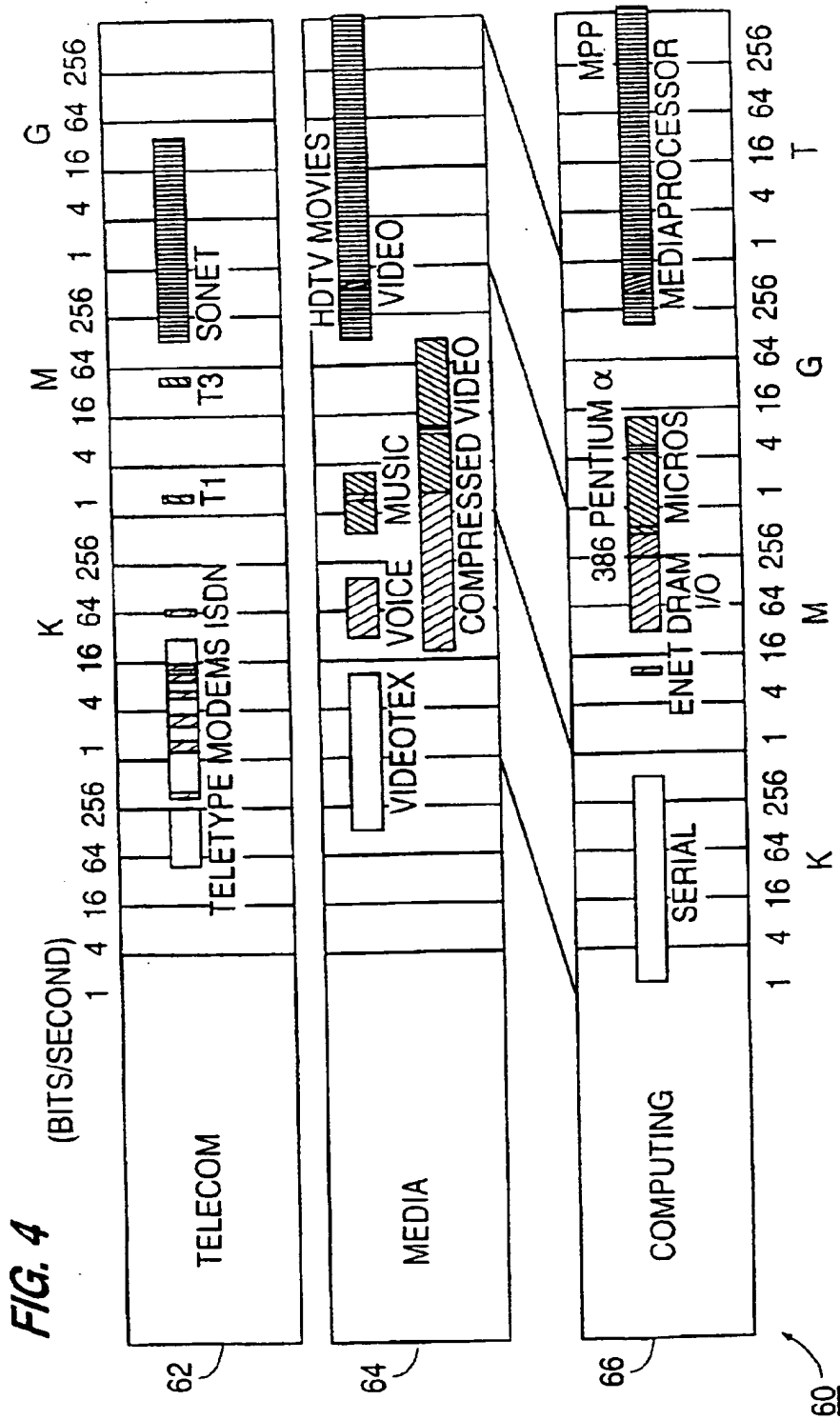
U.S. Patent

Aug. 11, 1998

Sheet 3 of 25

5,794,061





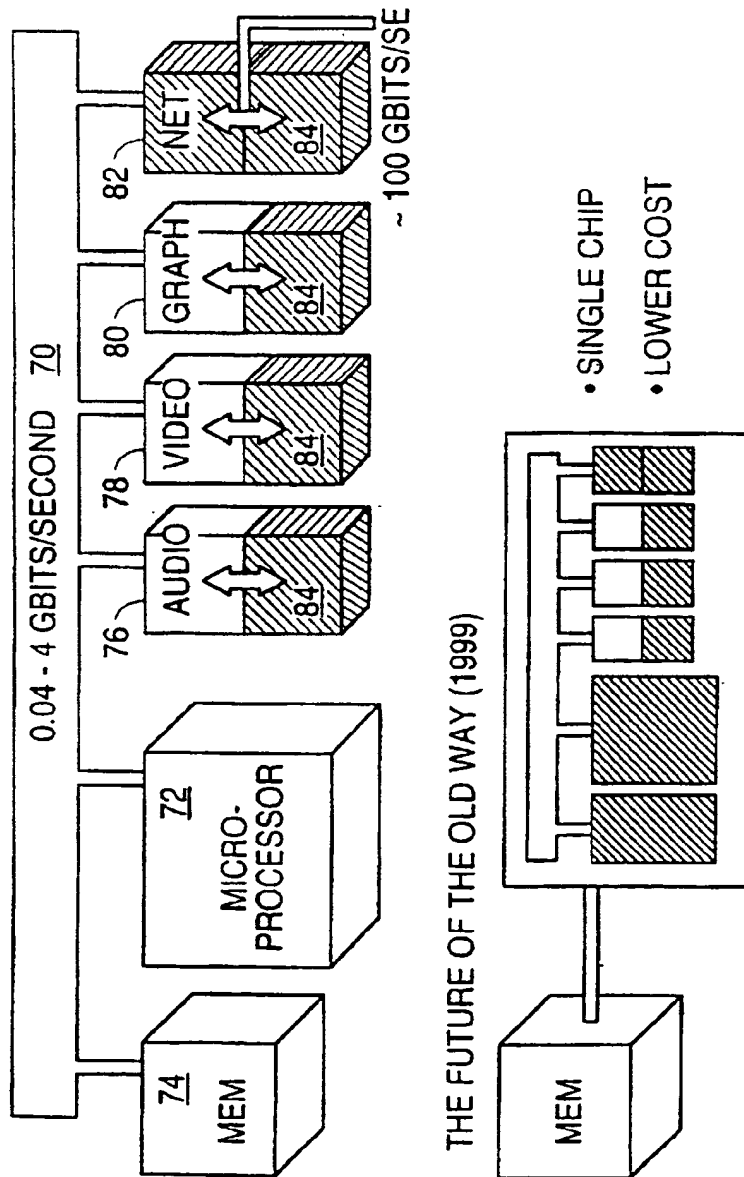
U.S. Patent

Aug. 11, 1998

Sheet 5 of 25

5,794,061

FIG. 5



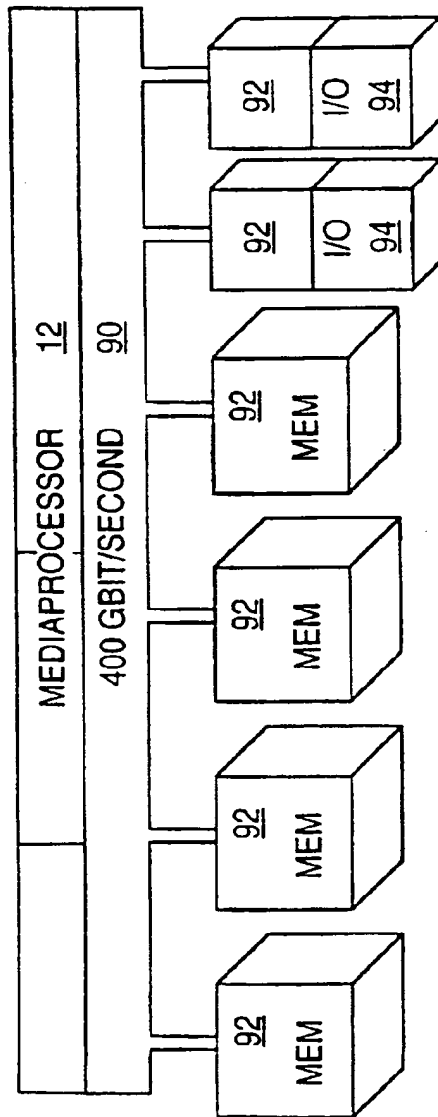
U.S. Patent

Aug. 11, 1998

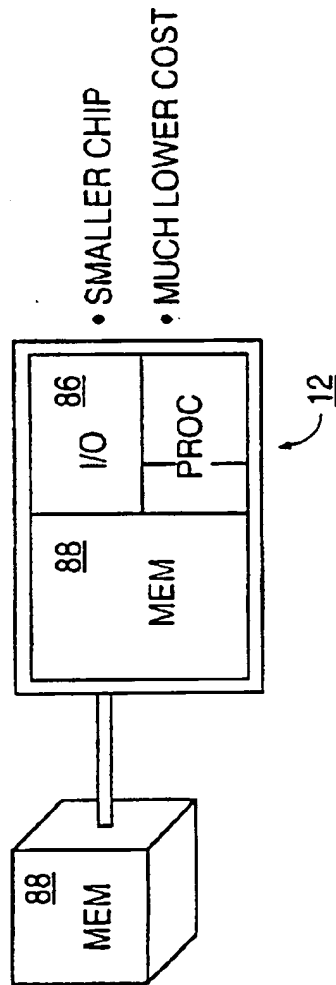
Sheet 6 of 25

5,794,061

FIG. 6



THE FUTURE OF THE UNIFIED WAY (1995)



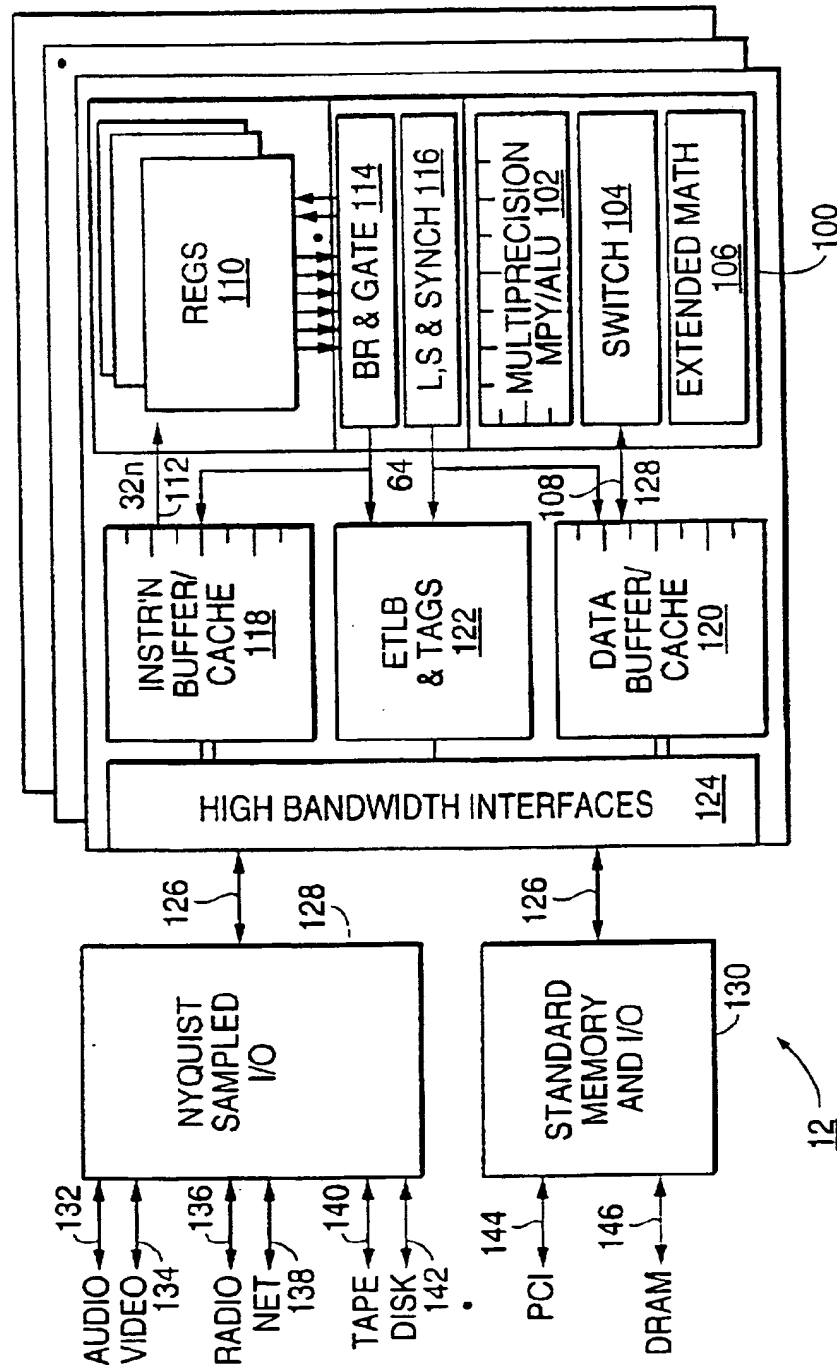
U.S. Patent

Aug. 11, 1998

Sheet 7 of 25

5,794,061

FIG. 7



U.S. Patent

Aug. 11, 1998

Sheet 8 of 25

5,794,061

FIG. 8(a)

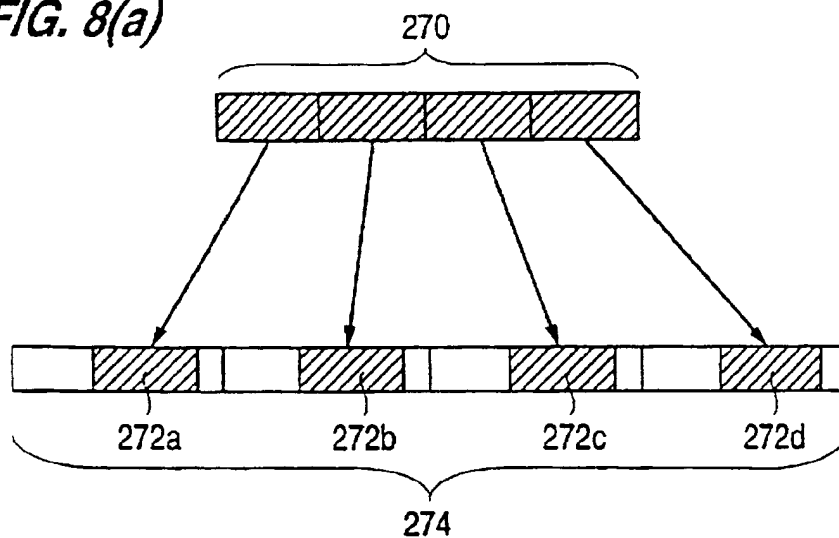
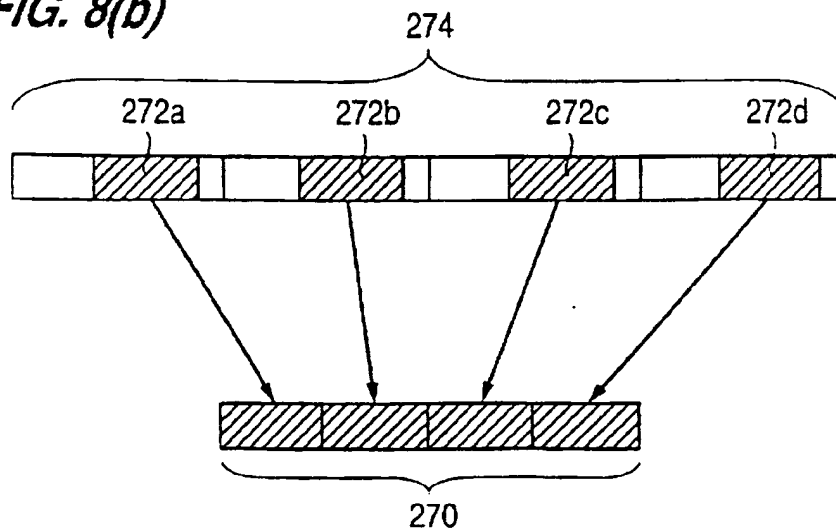


FIG. 8(b)



U.S. Patent

Aug. 11, 1998

Sheet 9 of 25

5,794,061

FIG. 8(c)

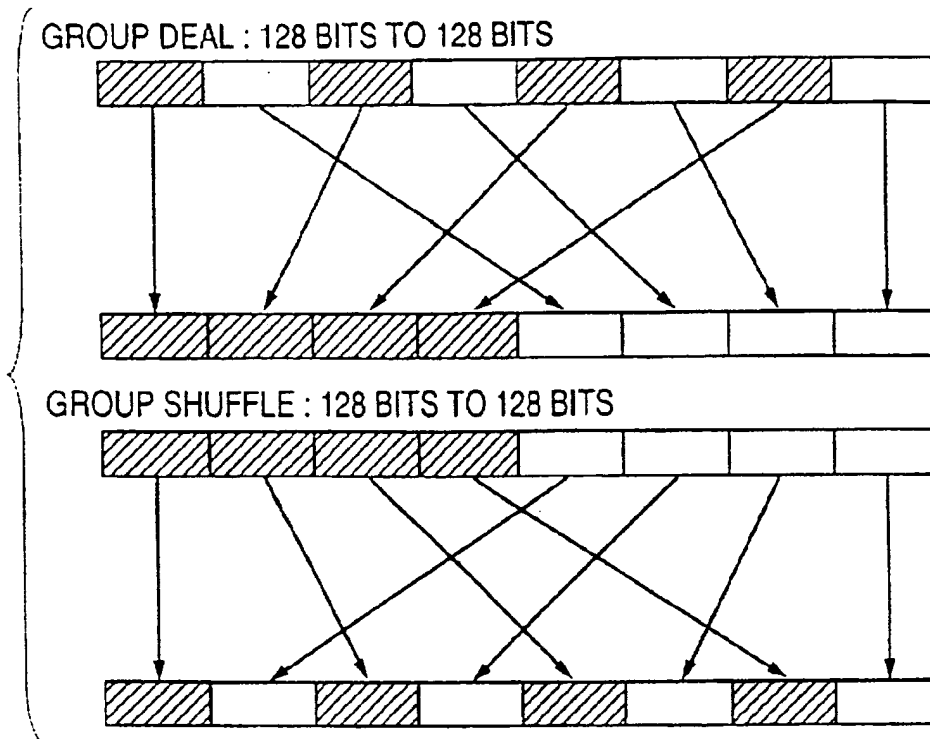
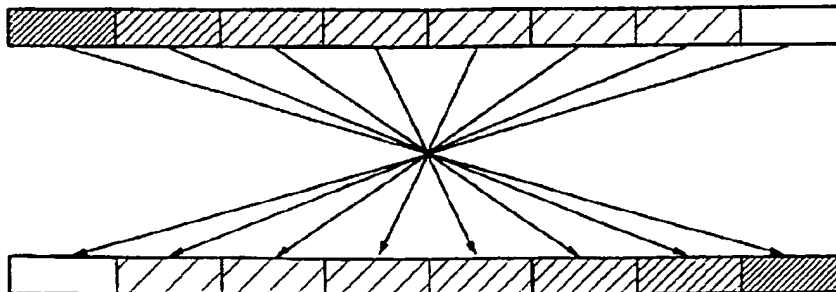


FIG. 8(d)



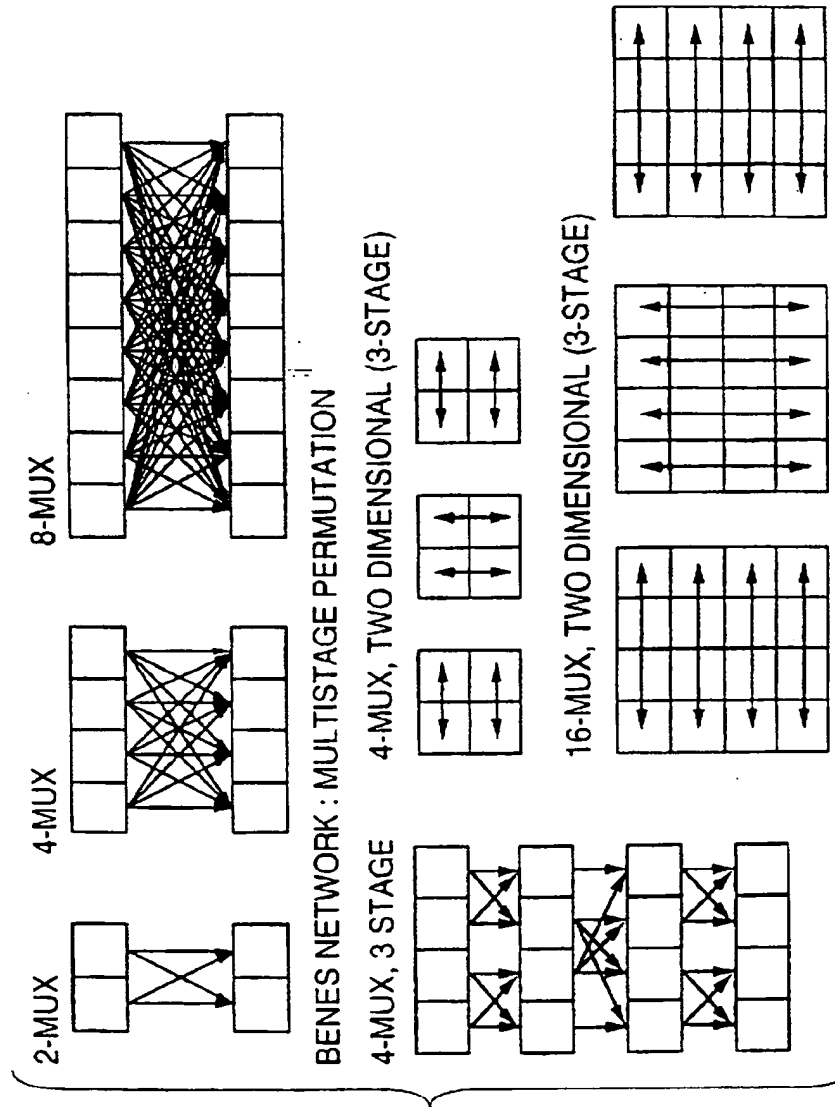
U.S. Patent

Aug. 11, 1998

Sheet 10 of 25

5,794,061

FIG. 8(e)



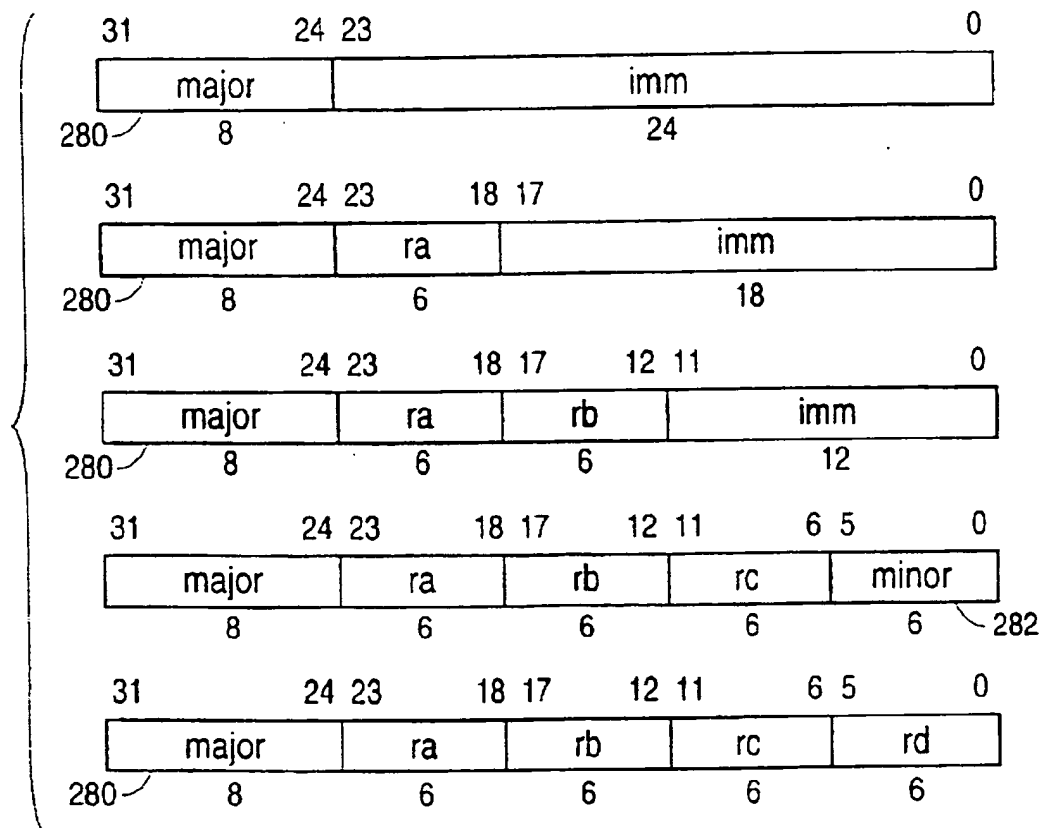
U.S. Patent

Aug. 11, 1998

Sheet 11 of 25

5,794,061

FIG. 9(a)



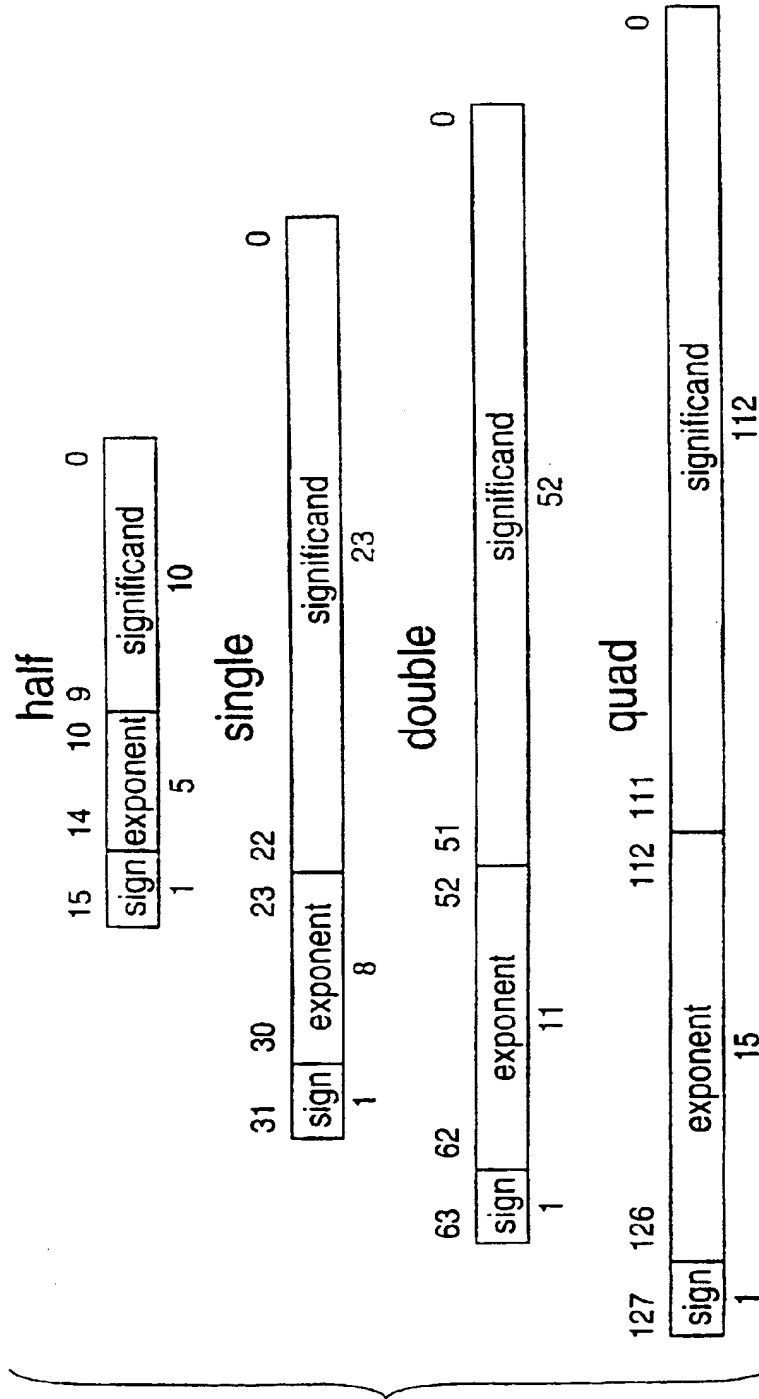
U.S. Patent

Aug. 11, 1998

Sheet 12 of 25

5,794,061

FIG. 9(b)



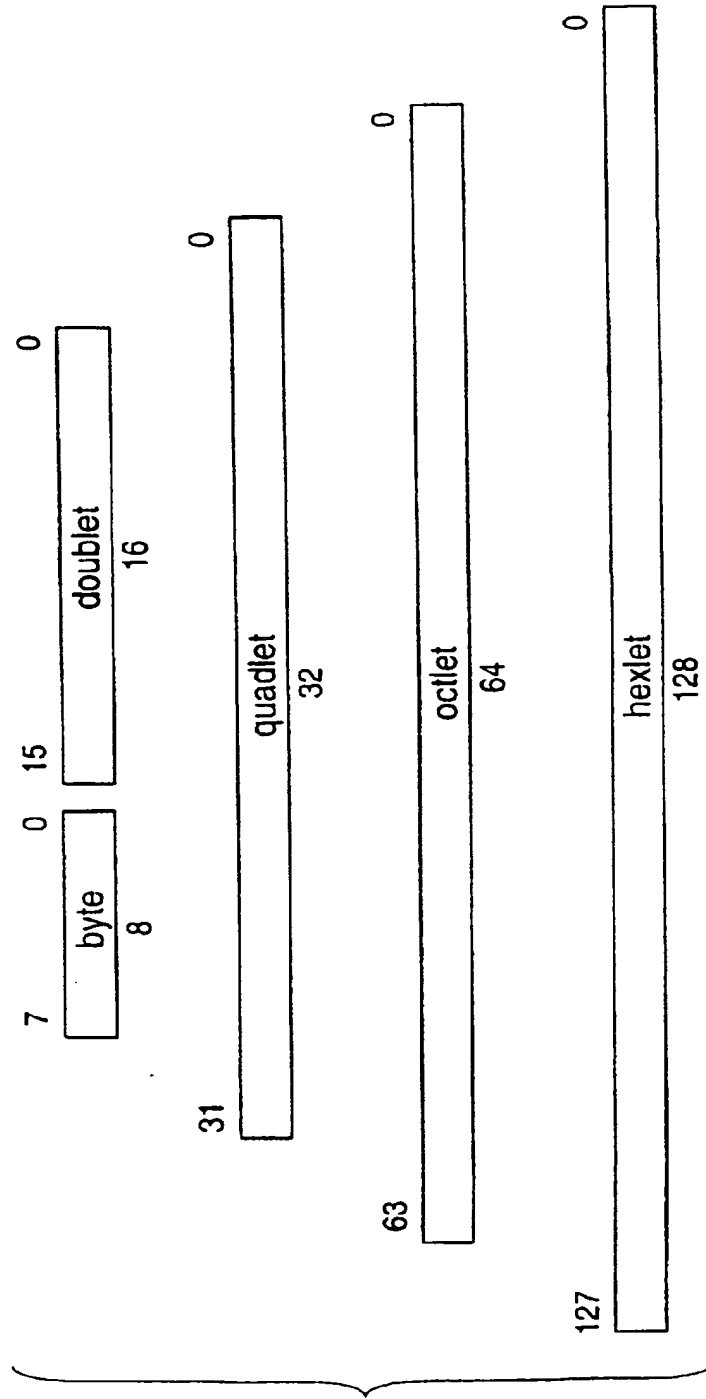
U.S. Patent

Aug. 11, 1998

Sheet 13 of 25

5,794,061

FIG. 9(c)



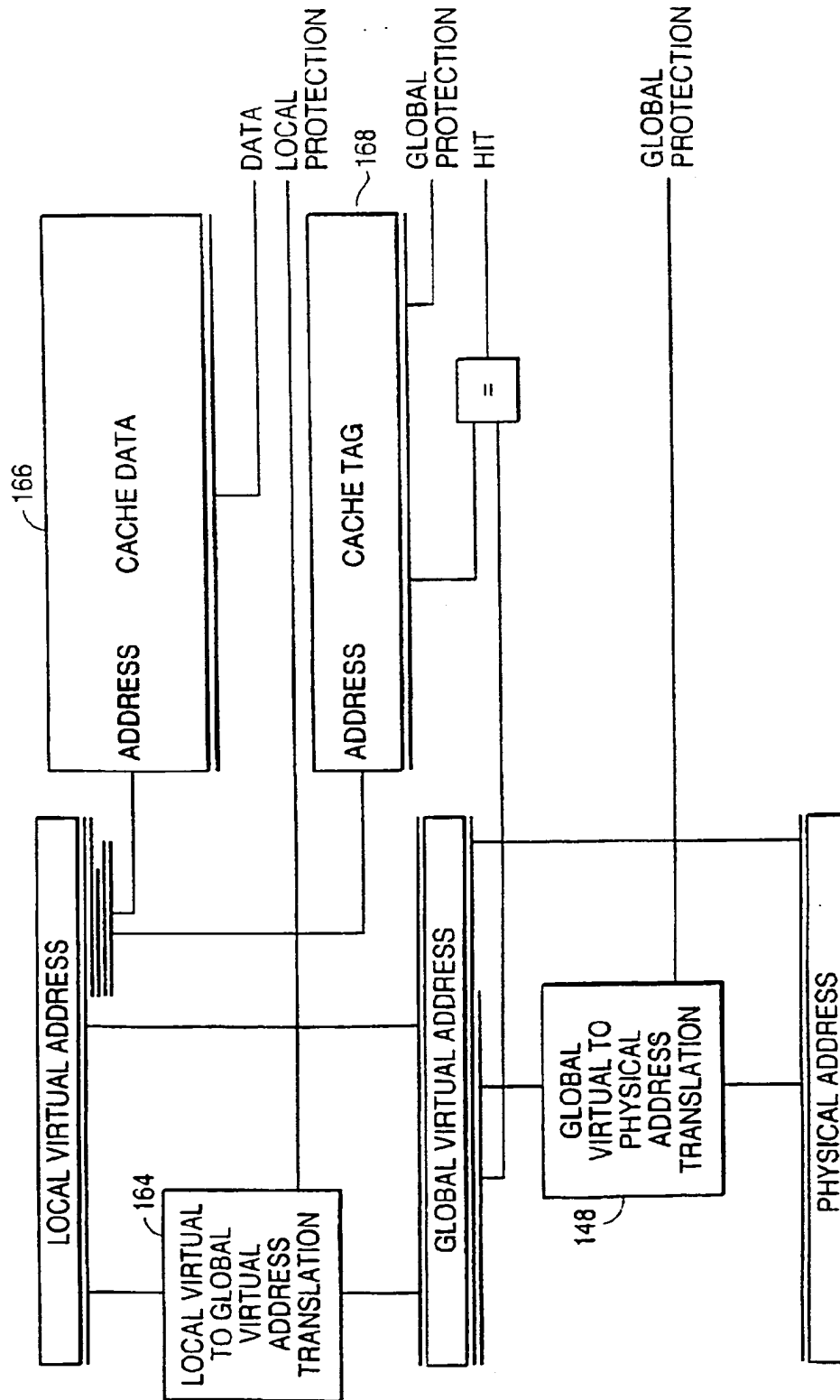
U.S. Patent

Aug. 11, 1998

Sheet 14 of 25

5,794,061

FIG. 10(a)



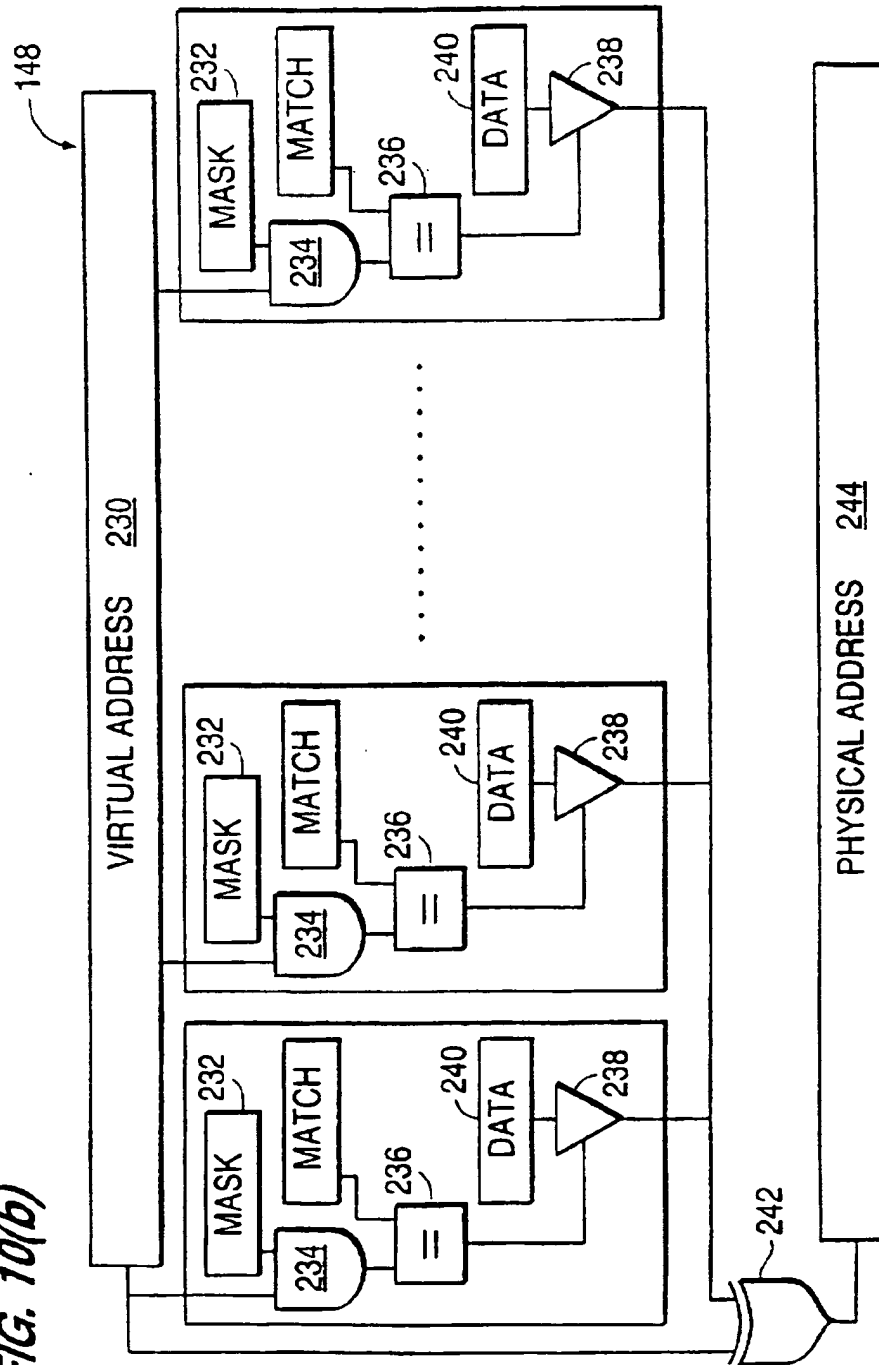
U.S. Patent

Aug. 11, 1998

Sheet 15 of 25

5,794,061

FIG. 10(b)



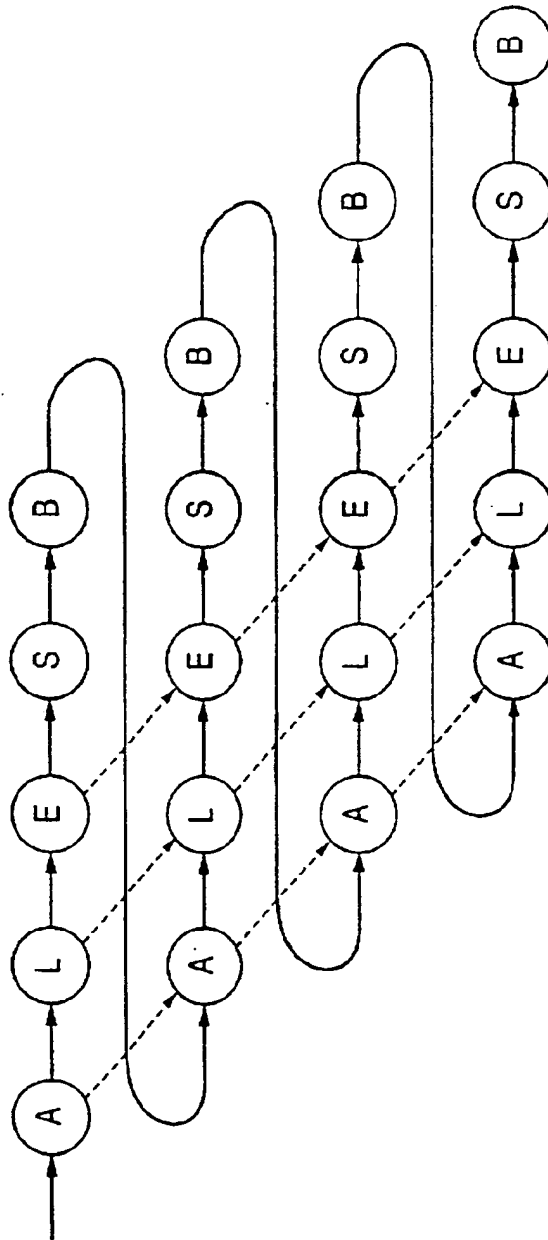
U.S. Patent

Aug. 11, 1998

Sheet 16 of 25

5,794,061

FIG. 11

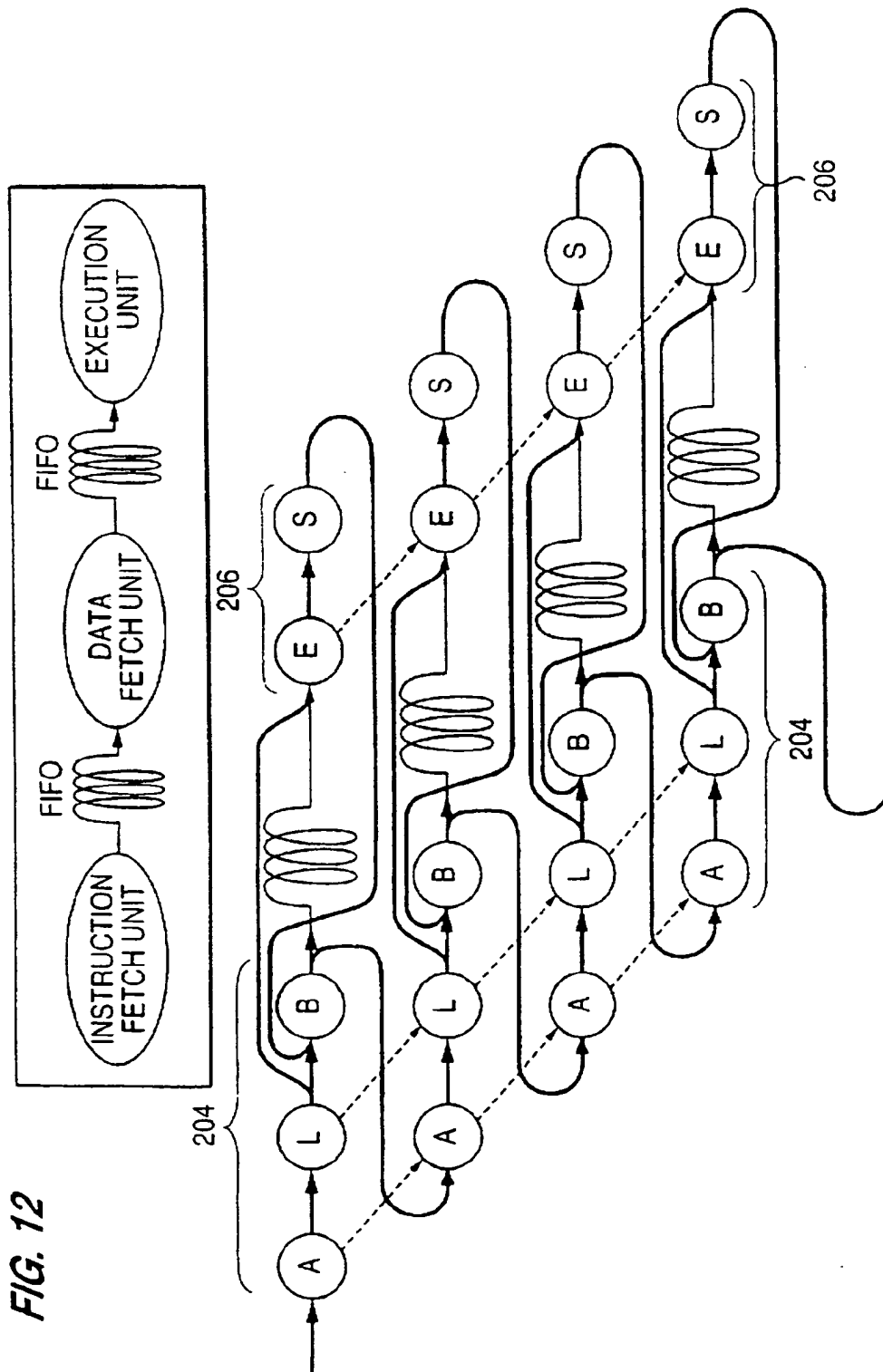


U.S. Patent

Aug. 11, 1998

Sheet 17 of 25

5,794,061



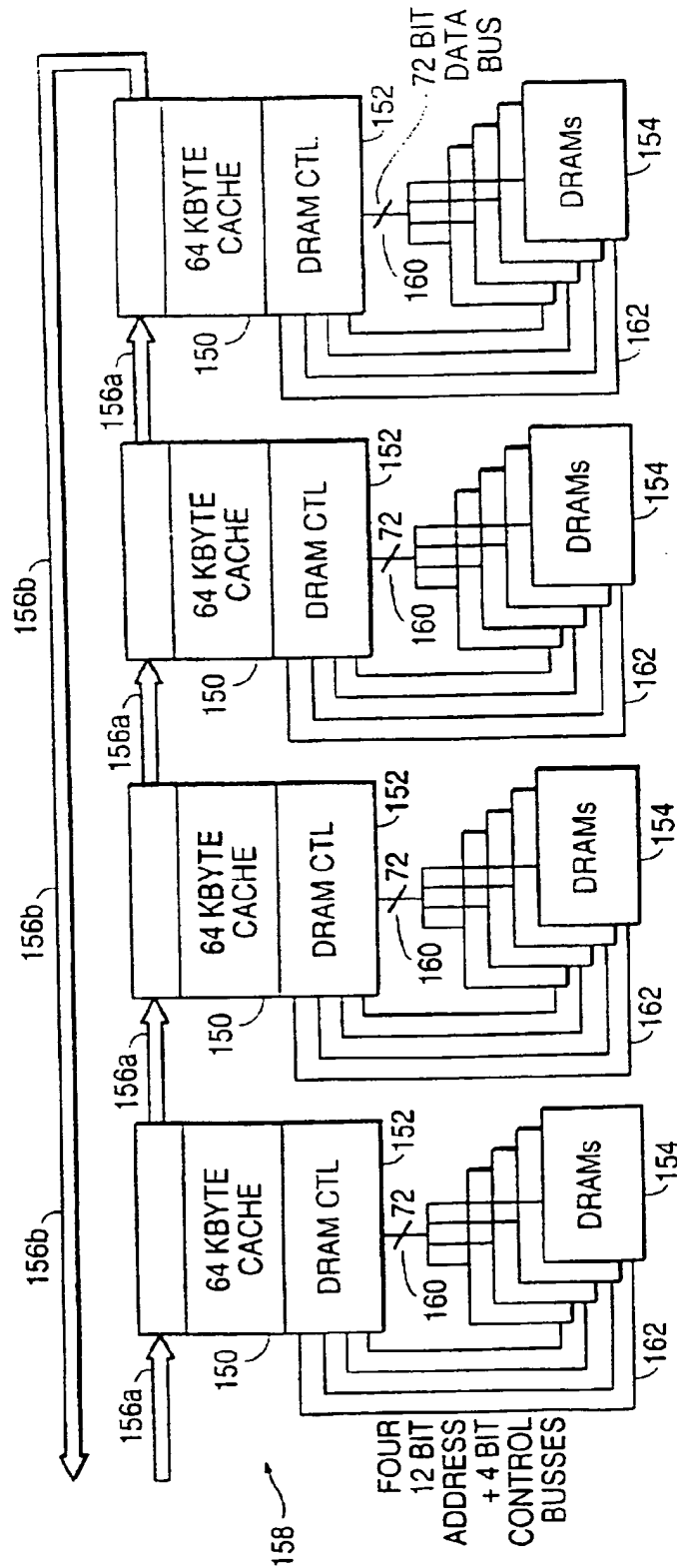
U.S. Patent

Aug. 11, 1998

Sheet 18 of 25

5,794,061

FIG. 13

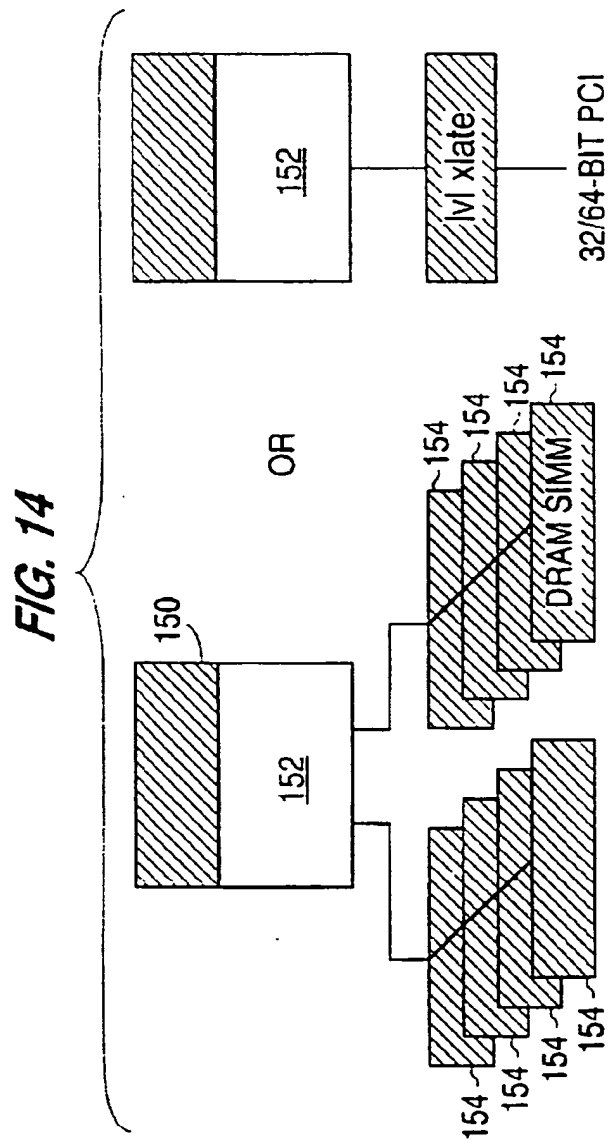


U.S. Patent

Aug. 11, 1998

Sheet 19 of 25

5,794,061



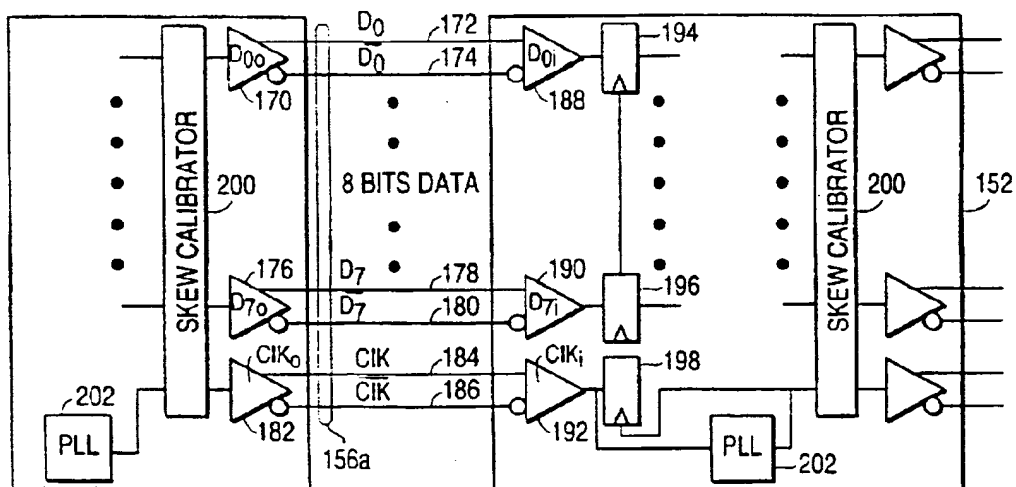
U.S. Patent

Aug. 11, 1998

Sheet 20 of 25

5,794,061

FIG. 15



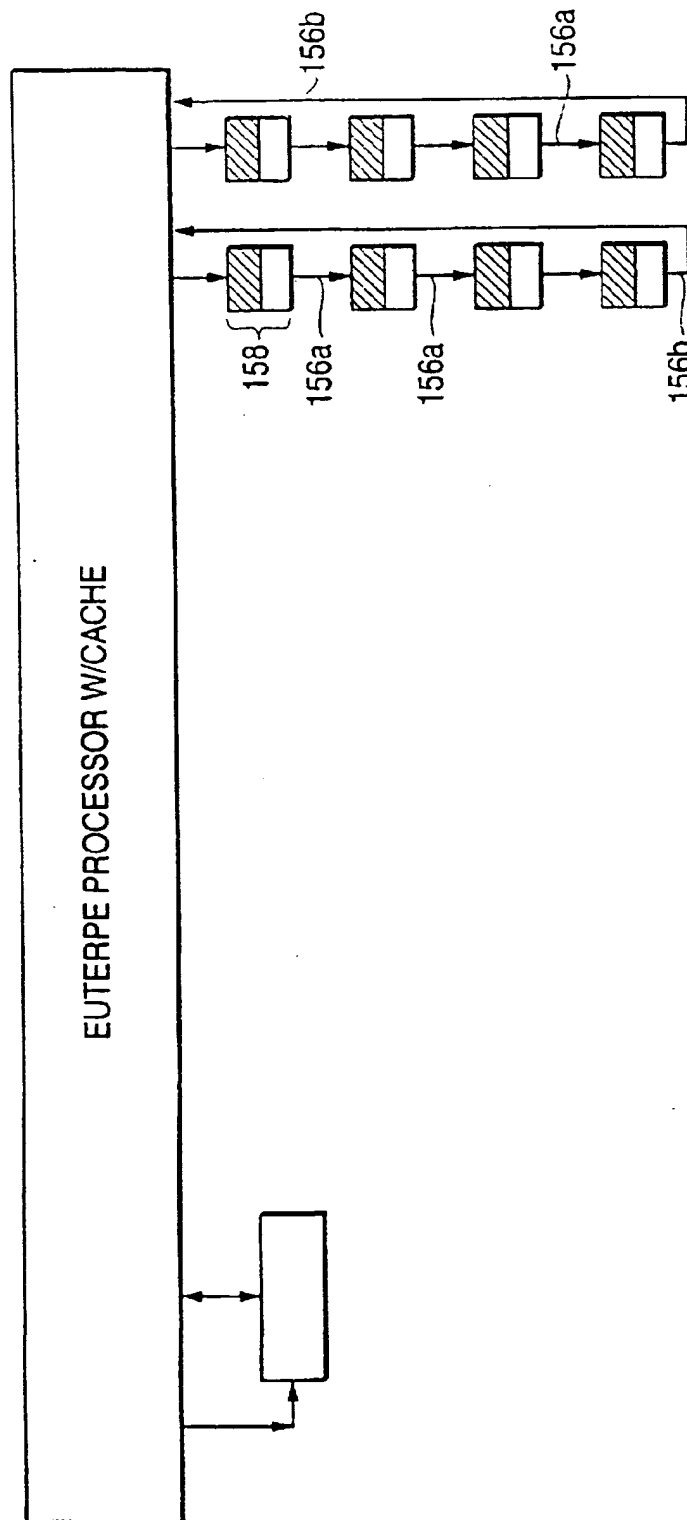
U.S. Patent

Aug. 11, 1998

Sheet 21 of 25

5,794,061

FIG. 16(a)



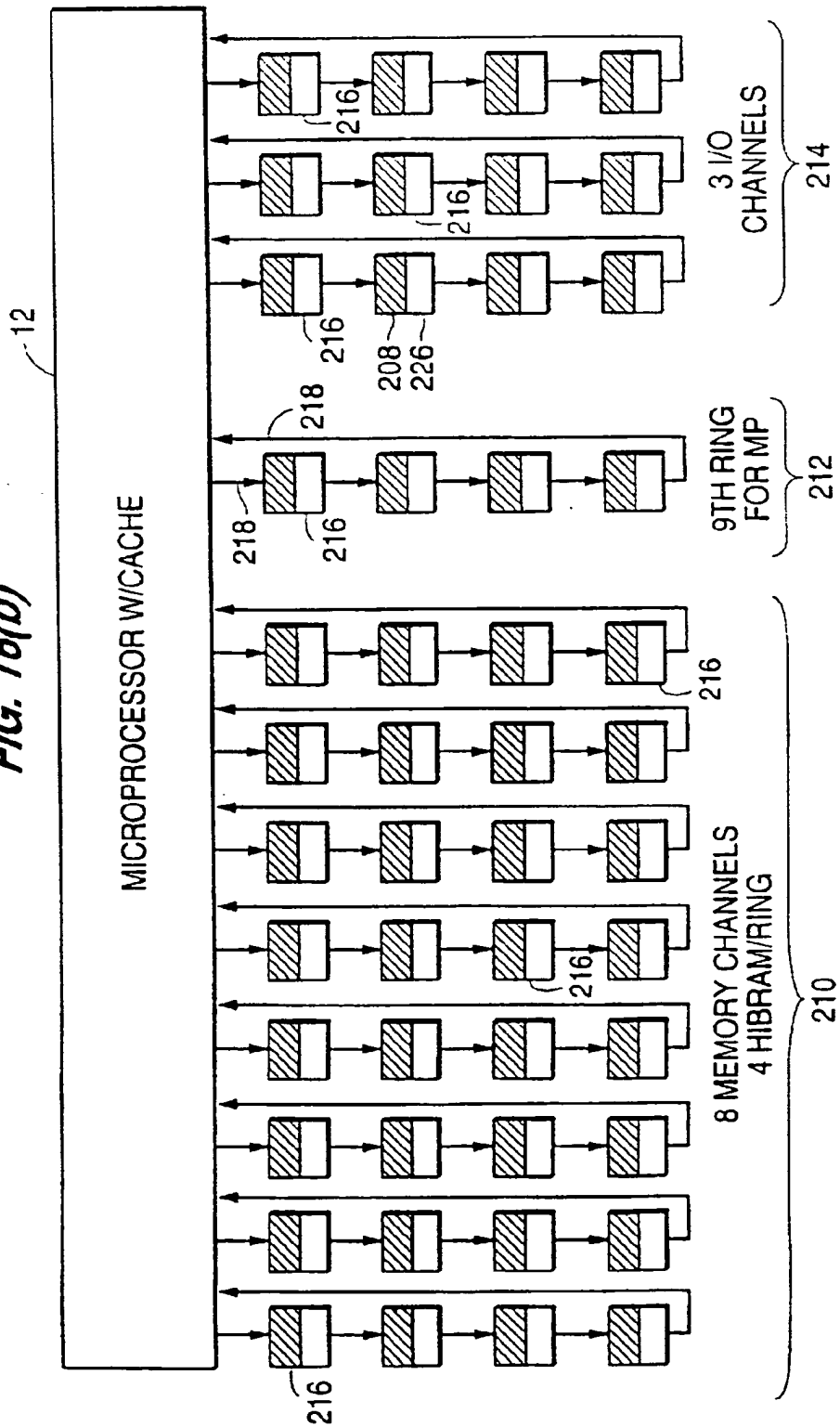
U.S. Patent

Aug. 11, 1998

Sheet 22 of 25

5,794,061

FIG. 16(b)

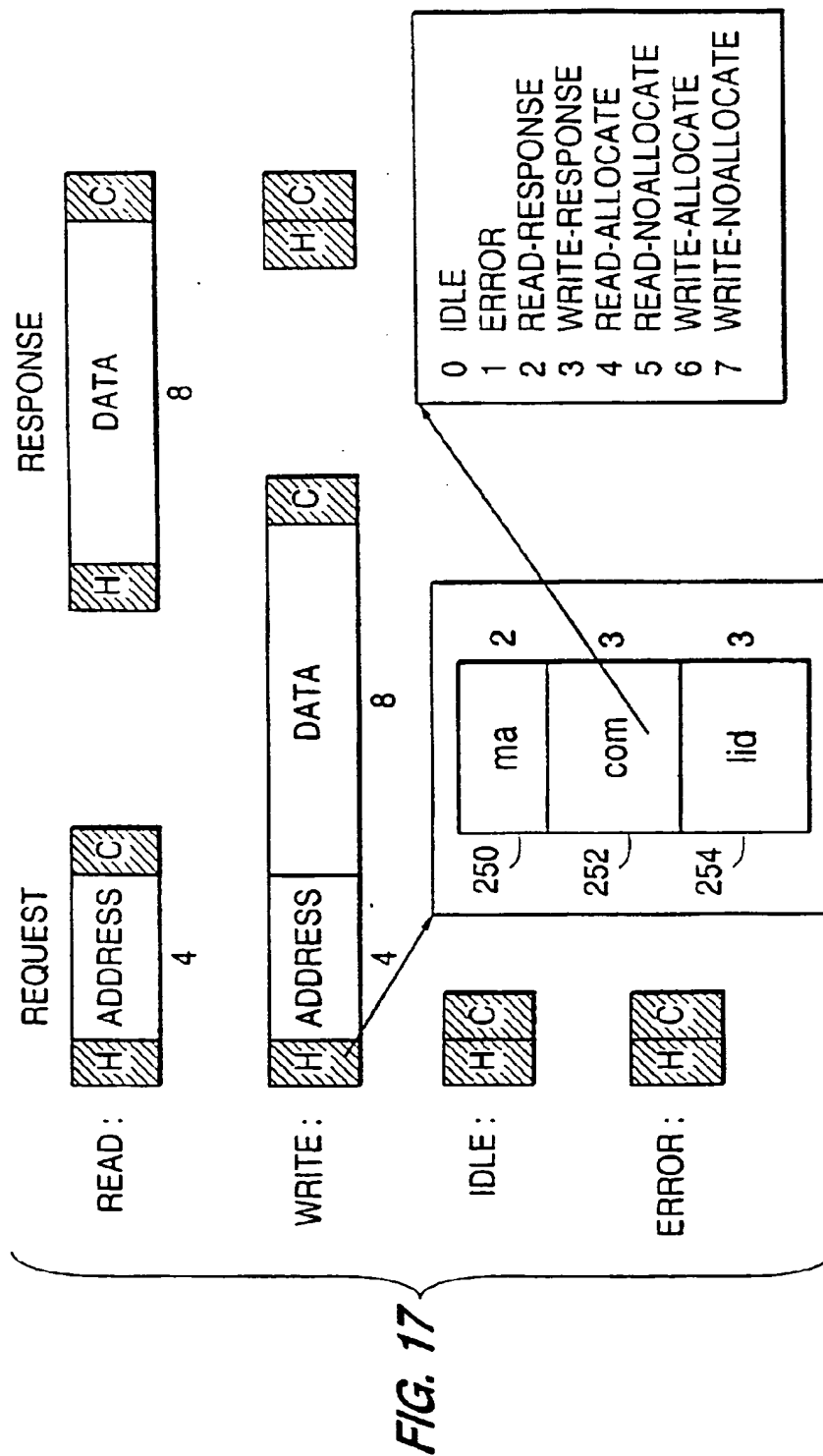


U.S. Patent

Aug. 11, 1998

Sheet 23 of 25

5,794,061



U.S. Patent

Aug. 11, 1998

Sheet 24 of 25

5,794,061

FIG. 18(a)

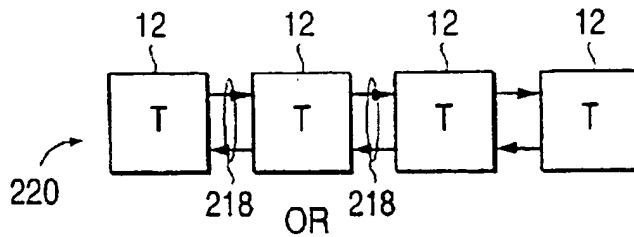


FIG. 18(b)

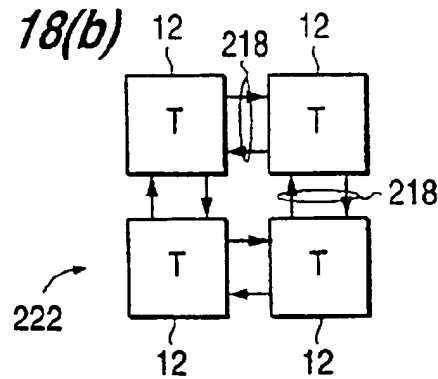
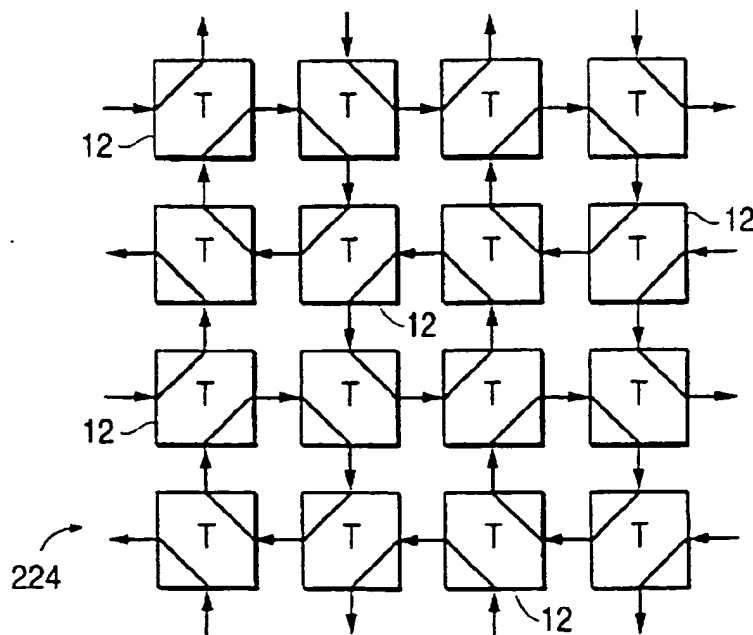


FIG. 18(c)



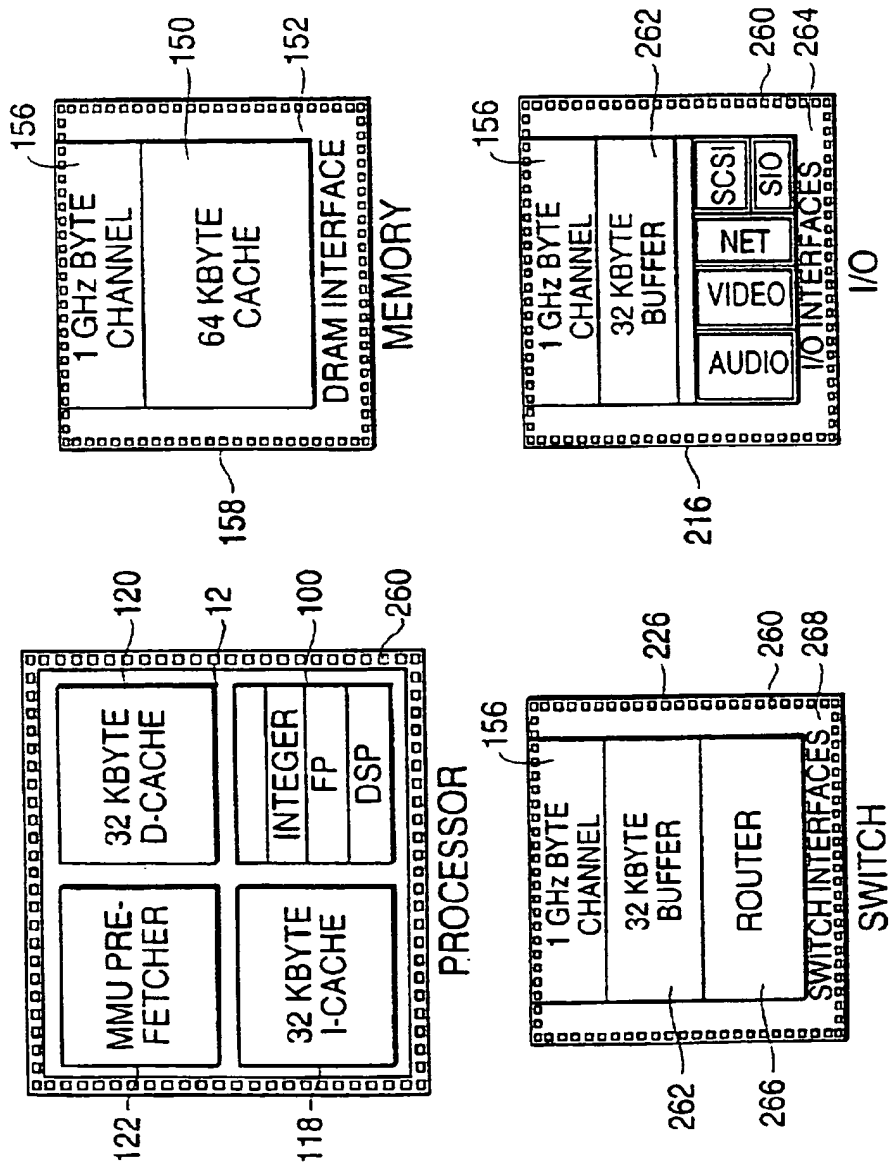
U.S. Patent

Aug. 11, 1998

Sheet 25 of 25

5,794,061

FIG. 19



5,794.061

1

GENERAL PURPOSE, MULTIPLE PRECISION PARALLEL OPERATION, PROGRAMMABLE MEDIA PROCESSOR

This is a divisional of application Ser. No. 08/516,036, filed Aug. 16, 1995, now U.S. Pat. No. 5,742,840.

A Microfiche Appendix consisting of 4 sheets (387 total frames) of microfiche is included in this application. The Microfiche Appendix contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by any one of the Microfiche Appendix, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

FIELD OF THE INVENTION

This invention relates to the field of communications processing, and more particularly, to a method and apparatus for real-time processing of multi-media digital communications.

BACKGROUND OF THE INVENTION

Optical fiber and discs have made the transmission and storage of digital information both cheaper and easier than older analog technologies. An improved system for digital processing of media data streams is necessary in order to realize the full potential of these advanced media.

For the past century, telephone service delivered over copper twisted pair has been the lingua franca of communications. Over the next century, broadband services delivered over optical fiber and coax will more completely fulfill the human need for sensory information by supplying voice, video, and data at rates of about 1,000 times greater than narrow band telephony. Current general-purpose microprocessors and digital signal processors ("DSPs") can handle digital voice, data, and images at narrow band rates, but they are way too slow for processing media data at broadband rates.

This shortfall in digital processing of broadband media is currently being addressed through the design of many different kinds of application-specific integrated circuits ("ASICs"). For example, a prototypical broadband device such as a cable modem modulates and demodulates digital data at rates up to 45 Mbits/sec within a single 6 MHz cable channel (as compared to rates of 28.8 Kbits/sec within a 6 KHz channel for telephone modems) and transcodes it onto a 10/100baseT connection to a personal computer ("PC") or workstation. Current cable modems thus receive data from a coaxial cable connection through a chain of specialized ASIC devices in order to accomplish Quadrature Amplitude Modulation ("QAM") demodulation, Reed-Solomon error correction, packet filtering, Data Encryption Standard ("DES") decryption, and Ethernet protocol handling. The cable modems also transmit data to the coaxial cable link through a second chain of devices to achieve DES encryption, Reed-Solomon block encoding, and Quaternary Phase Shift Keying ("QPSK") modulation. In these environments, a general-purpose processor is usually required as well in order to perform initialization, statistics collection, diagnostics, and network management functions.

The ASIC approach to media processing has three fundamental flaws: cost, complexity, and rigidity. The combined silicon area of all the specialized ASIC devices required in the cable modem, for example, results in a component cost incompatible with the per subscriber price target for a cable service. The cable plant itself is a very

2

hostile service environment, with noise ingress, reflections, nonlinear amplifiers, and other channel impairments, especially when viewed in the upstream direction. Telephony modems have developed an elaborate hierarchy of algorithms implemented in DSP software, with automatic reduction of data rates from 28.8Kbits/sec to 19.6Kbits/sec, 14.4Kbits/sec, or much lower rates as needed to accommodate noise, echoes, and other impairments in the copper plant. To implement similar algorithms on an ASIC-based broadband modem is far more complex to achieve in software.

These problems of cost, complexity, and rigidity are compounded further in more complete broadband devices such as digital set-top boxes, multimedia PCs, or video conferencing equipment, all of which go beyond the basic radio frequency ("RF") modem functions to include a broad range of audio and video compression and decoding algorithms, along with remote control and graphical user interfaces. Software for these devices must control what amounts to a heterogeneous multi-processor, where each specialized processor has a different, and usually eccentric or primitive, programming environment. Even if these programming environments are mastered, the degree of programmability is limited. For example, Motion Picture Expert Group-I ("MPEG-I") chips manufactured by AT&T Corporation will not implement advances such as fractal- and wavelet-based compression algorithms, but these chips are not readily software upgradable to the MPEG-II standard. A broadband network operator who leases an MPEG ASIC-based product is therefore at risk of having to continuously upgrade his system by purchasing significant amounts of new hardware just to track the evolution of MPEG standards.

The high cost of ASIC-based media processing results from inefficiencies in both memory and logic. A typical ASIC consists of a multiplicity of specialized logic blocks, each with a small memory dedicated to holding the data which comprises the working set for that block. The silicon area of these multiple small memories is further increased by the overhead of multiple decoders, sense amplifiers, write drivers, etc. required for each logic block. The logic blocks are also constrained to operate at frequencies determined by the internal symbol rates of broadband algorithms in order to avoid additional buffer memories. These frequencies typically differ from the optimum speed-area operating point of a given semiconductor technology. Interconnect and synchronization of the many logic and memory blocks are also major sources of overhead in the ASIC approach.

The disadvantages of the prior ASIC approach can be overcome by a single unified media processor. The cost advantages of such a unified processor can be achieved by gathering all the many ASIC functions of a broadband media product into a single integrated circuit. Cost reduction is further increased by reducing the total memory area of such a circuit by replacing the multiplicity of small ASIC memories with a single memory hierarchy large enough to accommodate the sum total of all the working sets, and wide enough to supply the aggregate bandwidth needs of all the logic blocks. Additionally, the logic block interconnect circuitry to this memory hierarchy may be streamlined by providing a generally programmable switching fabric. Many of the logic blocks themselves can also be replaced with a single multi-precision arithmetic unit, which can be internally partitioned under software control to perform addition, multiplication, division, and other integer and floating point arithmetic operations on symbol streams of varying widths, while sustaining the full data throughput of the memory

5.794.061

3

hierarchy. The residue of logic blocks that perform operations that are neither arithmetic or permutation group oriented can be replaced with an extended math unit that supports additional arithmetic operations such as finite field, ring, and table lookup, while also sustaining the full data throughput of the memory hierarchy.

The above multi-precision arithmetic, permutation switch, and extended math operations can then be organized as machine instructions that transfer their operands to and from a single wide multi-ported register file. These instructions can be further supplemented with load/store instructions that transfer register data to and from a data buffer/cache static random access memory ("SRAM") and main memory dynamic random access memories ("DRAMs"), and with branch instructions that control the flow of instructions executed from an instruction buffer/cache SRAM. Extensions to the load/store instructions can be made for synchronization, and to branch instructions for protected gateways, so that multiple threads of execution for audio, video, radio, encryption, networking, etc. can efficiently and securely share memory and logic resources of a unified machine operating near the optimum speed-area point of the target semiconductor process. The data path for such a unified media processor can interface to a high speed input/output ("I/O") subsystem that moves media streams across ultra-high bandwidth interfaces to external storage and I/O.

Such a device would incorporate all of the processing capabilities of the specialized multi-ASIC combination into a single, unified processing device. The unified processor would be agile and capable of reprogramming through the transmission of new programs over the communication medium. This programmable, general purpose device is thus less costly than the specialized processor combination, easier to operate and reprogram and can be installed or applied in many differing devices and situations. The device may also be scalable to communications applications that support vast numbers of users through massively parallel distributed computing.

It is therefore an object of this invention to process media data streams by executing operations at very high bandwidth rates.

It is also an object of this invention to unify the audio, video, radio, graphics, encryption, authentication, and networking protocols into a single instruction stream.

It is also an object of this invention to achieve high bandwidth rates in a unified processor that is easy to program and more flexible than a heterogeneous combination of special purpose processors.

It is a further object of the invention to support high level mathematical processing in a unified media processor, including finite group, finite field, finite ring and table look-up operations, all at high bandwidth rates.

It is yet a further object of the invention to provide a unified media processor that can be replicated into a multi-processor system to support a vast array of users.

It is yet another object of this invention to allow for massively parallel systems within the switching fabric to support very large numbers of subscribers and services.

It is also an object of the invention to provide a general purpose programmable processor that could be employed at all points in a network.

It is a further object of this invention to sustain very high bandwidth rates to arbitrarily large memory and input/output systems.

4

SUMMARY OF THE INVENTION

In view of the above, there is provided a system for media processing that maintains substantially peak data throughput in the execution and transmission of multiple media data streams. The system includes in one aspect a general purpose, programmable media processor, and in another aspect includes a method for receiving, processing and transmitting media data streams. The general purpose, programmable media processor of the invention further includes an execution unit, high bandwidth external interface, and can be employed in a parallel multi-processor system.

According to the apparatus of the invention, an execution unit is provided that maintains substantially peak data throughput in the unified execution of multiple media data streams. The execution unit includes a data path, and a multi-precision arithmetic unit coupled to the data path and capable of dynamic partitioning based on the elemental width of data received from the data path. The execution unit also includes a switch coupled to the data path that is programmable to manipulate data received from the data path and provide data streams to the data path. An extended mathematical element is also provided, which is coupled to the data path and programmable to implement additional mathematical operations at substantially peak data throughput. In a preferred embodiment of the execution unit, at least one register file is coupled to the data path.

According to another aspect of the invention, a general purpose programmable media processor is provided having an instruction path and a data path to digitally process a plurality of media data streams. The media processor includes a high bandwidth external interface operable to receive a plurality of data of various sizes from an external source and communicate the received data over the data path at a rate that maintains substantially peak operation of the media processor. At least one register file is included, which is configurable to receive and store data from the data path and to communicate the stored data to the data path. A multi-precision execution unit is coupled to the data path and is dynamically configurable to partition data received from the data path to account for the elemental symbol size of the plurality of media streams, and is programmable to operate on the data to generate a unified symbol output to the data path.

According to the preferred embodiment of the media processor, means are included for moving data between registers and memory by performing load and store operations, and for coordinating the sharing of data among a plurality of tasks by performing synchronization operations based upon instructions and data received by the execution unit. Means are also provided for securely controlling the sequence of execution by performing branch and gateway operations based upon instructions and data received by the execution unit. A memory management unit operable to retrieve data and instructions for timely and secure communication over the data path and instruction path respectively is also preferably included in the media processor. The preferred embodiment also includes a combined instruction cache and buffer that is dynamically allocated between cache space and buffer space to ensure real-time execution of multiple media instruction streams, and a combined data cache and buffer that is dynamically allocated between cache space and buffer space to ensure real-time response for multiple media data streams.

In another aspect of the invention, a high bandwidth processor interface for receiving and transmitting a media

5.794.061

5

stream is provided having a data path operable to transmit media information at sustained peak rates. The high bandwidth processor interface includes a plurality of memory controllers coupled in series to communicate stored media information to and from the data path, and a plurality of memory elements coupled in parallel to each of the plurality of memory controllers for storing and retrieving the media information. In the preferred embodiment of the high bandwidth processor interface, the plurality of memory controllers each comprise a paired link disposed between each memory controller, where the paired links each transmit and receive plural bits of data and have differential data inputs and outputs and a differential clock signal.

Yet another aspect of the invention includes a system for unified media processing having a plurality of general purpose media processors, where each media processor is operable at substantially peak data rates and has a dynamically partitioned execution unit and a high bandwidth interface for communicating to memory and input/output elements to supply data to the media processor at substantially peak rates. A bi-directional communication fabric is provided, to which the plurality of media processors are coupled, to transmit and receive at least one media stream comprising presentation, transmission, and storage media information. The bi-directional communication fabric preferably comprises a fiber optic network, and a subset of the plurality of media processors comprise network servers.

According to yet another aspect of the invention, a parallel multimedia processor system is provided having a data path and a high bandwidth external interface coupled to the data path and operable to receive a plurality of data of various sizes from an external source and communicate the received data at a rate that maintains substantially peak operation of the parallel multi-processor system. A plurality of register files, each having at least one register coupled to the data path and operable to store data, are also included. At least one multi-precision execution unit is coupled to the data path and is dynamically configurable to partition data received from the data path to account for the elemental symbol size of the plurality of media streams, and is programmable to operate in parallel on data stored in the plurality of register files to generate a unified symbol output for each register file.

According to the method of the invention, unified streams of media data are processed by receiving a stream of unified media data including presentation, transmission and storage information. The unified stream of media data is dynamically partitioned into component fields of at least one bit based on the elemental symbol size of data received. The unified stream of media data is then processed at substantially peak operation.

In one aspect of the invention, the unified stream of media data is processed by storing the stream of unified media data in a general register file. Multi-precision arithmetic operations can then be performed on the stored stream of unified media data based on programmed instructions, where the multi-precision arithmetic operations include Boolean, integer and floating point mathematical operations. The component fields of unified media data can then be manipulated based on programmed instructions that implement copying, shifting and re-sizing operations. Multi-precision mathematical operations can also be performed on the stored stream of unified media data based on programmed instructions, where the mathematical operations including finite group, finite field, finite ring and table look-up operations. Instruction and data pre-fetching are included to fill instruction and data pipelines, and memory management

6

operations can be performed to retrieve instructions and data from external memory. The instructions and data are preferably stored in instruction and data cache/buffers, in which buffer storage in the instruction and data cache/buffers is dynamically allocated to ensure real-time execution.

Other aspects of the invention include a method for achieving high bandwidth communications between a general purpose media processor and external devices by providing a high bandwidth interface disposed between the media processor and the external devices, in which the high bandwidth interface comprises at least one uni-directional channel pair having an input port and an output port. A plurality of media data streams, comprising component fields of various sizes, are transmitted and received between the media processor and the external devices at a rate that sustains substantially peak data throughput at the media processor. A method for processing streams of media data is also included that provides a bi-directional communications fabric for transmitting and receiving at least one stream of media data, where the at least one stream of media data comprises presentation, transmission and storage information. At least one programmable media processor is provided within the communications network for receiving, processing and transmitting the at least one stream of unified media data over the bidirectional communications fabric.

The general purpose, programmable media processor of the invention combines in a single device all of the necessary hardware included in the specialized processor combinations to process and communicate digital media data streams in real-time. The general purpose, programmable media processor is therefore cheaper and more flexible than the prior approach to media processing. The general purpose, programmable media processor is thus more susceptible to incorporation within a massively parallel processing network of general purpose media processors that enhance the ability to provide real-time multi-media communications to the masses.

These features are accomplished by deploying server media processors and client media processors throughout the network. Such a network provides a seamless, global media super-computer which allows programmers and network owners to virtualize resources. Rather than restrictively accessing only the memory space and processing time of a local resource, the system allows access to resources throughout the network. In small access points such as wireless devices, where very little memory and processing logic is available due to limited battery life, the system is able to draw upon the resources of a homogeneous multi-computer system.

The invention also allows network owners the facility to track standards and to deploy new services by broadcasting software across the network rather than by instituting costly hardware upgrades across the whole network. Broadcasting software across the network can be performed at the end of an advertisement or other program that is broadcasted nationally. Thus, services can be advertised and then transmitted to new subscribers at the end of the advertisement.

These and other features and advantages of the invention will be apparent upon consideration of the following detailed description of the presently preferred embodiments of the invention, taken in conjunction with the appended drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a broad band media computer employing the general purpose, programmable media processor of the invention;

5.794,061

7

FIG. 2 is a block diagram of a global media processor employing multiple general purpose media processors according to the invention;

FIG. 3 is an illustration of the digital bandwidth spectrum for telecommunications, media and computing communications;

FIG. 4 is the digital bandwidth spectrum shown in FIG. 3 taking into account the bandwidth overhead associated with compressed video techniques;

FIG. 5 is a block diagram of the current specialized processor solution for mass media communication, where FIG. 5 shows the current distributed system, and shows a possible integrated approach;

FIG. 6 is a block diagram of two presently preferred general purpose media processors, where FIG. 6(a) shows a distributed system and shows an integrated media processor;

FIG. 7 is a block diagram of the presently preferred structure of a general purpose, programmable media processor according to the invention;

FIG. 8 is a drawing consisting of visual illustrations of the various group operations provided on the media processor, where FIG. 8(a) illustrates the group expand operation, FIG. 8(b) illustrates the group compress or extract operation, FIG. 8(c) illustrates the group deal and shuffle operations, FIG. 8(d) illustrates the group swizzle operation and FIG. 8(e) illustrates the various group permute operations;

FIG. 9 shows the preferred instruction and data sizes for the general purpose, programmable media processor, where FIG. 9(a) is an illustration of the various instruction formats available on the general purpose, programmable media processor, FIG. 9(b) illustrates the various floating-point data sizes available on the general purpose media processor, and FIG. 9(c) illustrates the various fixed-point data sizes available on the general purpose media processor;

FIG. 10 is an illustration of a presently preferred memory management unit included in the general purpose processor shown in FIG. 7, where FIG. 10(a) is a translation block diagram and FIG. 10(b) illustrates the functional blocks of the transaction lookaside buffer;

FIG. 11 is an illustration of a super-string pipeline technique;

FIG. 12 is an illustration of the presently preferred super-string pipeline technique;

FIG. 13 is a block diagram of a single memory channel for communication to the general purpose media processor shown in FIG. 7;

FIG. 14 is an illustration of the presently preferred connection of standard memory devices to the preferred memory interface;

FIG. 15 is a block diagram of the input/output controller for use with the memory channel shown in FIG. 13;

FIG. 16 is a block diagram showing multiple memory channels connected to the general purpose media processor shown in FIG. 7, where FIG. 16(a) shows a two-channel implementation and FIG. 16(b) illustrates a twelve-channel embodiment;

FIG. 17 illustrates the presently preferred packet communications protocol for use over the memory channel shown in FIG. 13;

FIG. 18 shows a multi-processor configuration employing the general purpose media processor shown in FIG. 7, where FIG. 18(a) shows a linear processor configuration, FIG. 18(b) shows a processor ring configuration, and FIG. 18(c) shows a two-dimensional processor configuration; and

8

FIG. 19 shows a presently preferred multi-chip implementation of the general purpose, programmable media processor of the invention.

DETAILED DESCRIPTION OF THE PRESENTLY PREFERRED EMBODIMENTS

Referring to the drawings, where like-reference numerals refer to like elements throughout, a broad band microcomputer 10 is provided in FIG. 1. The broad band microcomputer 10 consists essentially of a general purpose media processor 12. As will be described in more detail below, the general purpose media processor 12 receives, processes and transmits media data streams in a bidirectional manner from upstream network components to downstream devices. In general, media data streams received from upstream network components can comprise any combination of audio, video, radio, graphics, encryption, authentication, and networking information. As those skilled in the art will appreciate, however, the general purpose media processor 12 is in no way limited to receiving, processing and transmitting only these types of media information. The general purpose media processor 12 of the invention is capable of processing any form of digital media information without departing from the spirit and essential scope of the invention.

System Configuration

In the preferred embodiment of the invention shown in FIG. 1, media data streams are communicated to the media processor 12 from several sources. Ideally, unified media data streams are received and transmitted by the general purpose media processor 12 over a fiber optic cable network 14. As will be described in more detail below, although a fiber optic cable network is preferred, the presently existing communications network in the United States consists of a combination of fiber optic cable, coaxial cable and other transmission media. Consequently, the general purpose media processor 12 can also receive and transmit media data streams over coaxial cable 14 and traditional twisted pair wire connections 16. The specific communications protocol employed over the twisted pair 16, whether POTS, ISDN or ADSL, is not essential; all protocols are supported by the broad band microcomputer 10. The details of these protocols are generally known to those skilled in the art and no further discussion is therefore needed or provided herein.

Another form of upstream network communication is through a satellite link 18. The satellite link 18 is typically connected to a satellite receiver 20. The satellite receiver 20 comprises an antenna, usually in the form of a satellite dish, and amplification circuitry. The details of such satellite communications are also generally known in the art, and further detail is therefore not provided or included herein.

As described above, the general purpose media processor 12 communicates in a bidirectional manner to receive, process and transmit media data streams to and from downstream devices. As shown in FIG. 1, downstream communication preferably takes place in at least two forms. First, media data streams can be communicated over a bi-directional local network 22. Various types of local networks 22 are generally known in the art and many different forms exist. The general purpose media processor 12 is capable of communicating over any of these local networks 22 and the particular type of network selected is implementation specific.

The local network 22 is preferably employed to communicate between the unified processor 12, and audio/visual

5.794.061

9

devices 24 or other digital devices 26. Presently preferred examples of audio/visual devices 24 include digital cable television, video-on-demand devices, electronic yellow pages services, integrated message systems, video telephones, video games and electronic program guides. As those skilled in the art will appreciate, other forms of audio/video devices are contemplated within the spirit and scope of the invention. Presently preferred embodiments of other digital devices 26 for communication with the general purpose media processor 12 include personal computers, television sets, work stations, digital video camera recorders, and compact disc read-only memories. As those skilled in the art will also appreciate, further digital devices 26 are contemplated for communication to the general purpose media processor 12 without departing from the spirit and scope of the invention.

Second, the general purpose media processor preferably also communicates with downstream devices over a wireless network 28. In the presently preferred embodiment of the invention, wireless devices for communication over the wireless network 28 can comprise either remote communication devices 30 or remote computing devices 32. Presently preferred embodiments of the remote communications devices 30 include cordless telephones and personal communicators. Presently preferred embodiments of the remote computing devices 32 include remote controls and telecommunicating devices. As those skilled in the art will appreciate, other forms of remote communication devices 30 and remote computing devices 32 are capable of communication with the general purpose media processor 12 without departing from the spirit and scope of the invention. An agile digital radio (not shown) that incorporates a general purpose media processor 12 may be used to communicate with these wireless devices.

Network Configuration

Referring now to FIG. 2, the general purpose media processor 12 is preferably disposed throughout a digital communications network 38. In order to enable communication among large and small businesses, residential customers and mobile users, the network 38 can consist of a combination of many individual subnetworks comprised of three main forms of interconnection. The trunk and main branches of the network 38 preferably employ fiber optic cable 40 as the preferred means of interconnection. Fiber optic cable 40 is used to connect between general purpose media processors 12 disposed as network servers 46 or large business installations 48 that are capable of coupling directly to the fiber optic link 40. For communications to small business and residential customers that may be incapable of directly coupling to the fiber optic cable 40, a general purpose media processor 12 can be used as an interface to other forms of network interconnection.

As shown in FIG. 2, alternate forms of interconnection consist of coaxial cable lines 42 and twisted pair wiring 44. Coaxial cable lines are currently in place throughout the U.S. and is typically employed to provide cable television services to residential homes. According to the preferred embodiment of the invention, general purpose media processors 12 can be installed at these residential locations 52. In contrast to the specialized processor approach, the general purpose media processor 12 provides enough bandwidth to allow for bi-directional communications to and from these residential locations 52.

Network servers 46 controlled by general purpose media processors 12 are also employed throughout the network 38.

10

For example, the network servers 46 can be used to interface between the fiber optic network 40 and twisted pair wiring 44. Twisted pair wiring 44 is still employed for small businesses 50 and residential locations 52 that do not or cannot currently subscribe to coaxial cable or fiber optic network services. General purpose media processors 12 are also disposed at these small business locations 50 and non-cable residential locations 52. General purpose media processors 12 are also installed in wireless or mobile locations 52, which are coupled to the network 38 through agile digital radios (not shown). As shown in FIG. 2, network databases or other peripherals 56 can also be coupled to general purpose media processors 12 in the network 38.

The general purpose media processor 12 is operable at significantly high bandwidths in order to receive, process and transmit unified media data streams. Referring to FIG. 3, the respective frequencies for various types of media data streams are set forth against a bandwidth spectrum 60. The bandwidth spectrum 60 includes three component spectrums, all along the same range of frequencies, which represent the various frequency rates of digital media communications. Current computing bandwidth capabilities are also displayed. The telecommunications spectrum 62 shows the various frequency bands used for telecommunications transmission. For example, teletype terminals and modems operate in a range between approximately 64 kilobits/second to 16 kilobits/second. The ISDN telecommunication protocol operates at 64 kilobits/second. At the upper end of the telecommunications spectrum 62, T1 and T3 trunks operate at one megabit per second and 32 megabits per second, respectively. The SONET frequency range extends from approximately 128 megabits per second up to approximately 32 gigabits per second. Accordingly, in order to carry such broad band communications, the general purpose media processor 12 is capable of transferring information at rates into the gigabits per second range or higher.

A spectrum of typical media data streams is presented in the media spectrum 64 shown in FIG. 3. Voice and music transmissions are centered at frequencies of approximately 64 kilobits per second and one megabit per second, respectively. At the upper end of the media spectrum 64, video transmission takes place in a range from 128 megabits per second for high density television up to over 256 gigabits per second for movie applications. When using common video compression techniques, however, the video transmission spectrum can be shifted down to between 32 kilobits per second to 128 megabits per second as a result of the data compression. As described below, the processing required to achieve the data compression results in an increase in bandwidth requirements.

Current computing bandwidths are shown in the computing spectrum 66 of FIG. 3. Serial communications presently take place in a range between two kilobits per second up to 512 kilobits per second. The Ethernet network protocol operates at approximately 8 megabits per second. Current dynamic random access memory and other digital input/output peripherals operate between 32 megabits per second and 512 megabits per second. Presently available microprocessors are capable of operation in the low gigabits per second range. For example, the '386 Pentium microprocessor manufactured by Intel Corporation of Santa Clara, Calif. operates in the lower half of that range, and the Alpha microprocessor manufactured by Digital Equipment Corporation approaches the 16 gigabits per second range.

When video compression is employed, as expressed above, the associated processing overhead reduces the effective bandwidth of the particular processor. As a result, in

5,794,061

11

order to handle compressed video, these processors must operate in the terahertz frequency range. The bandwidth spectrum 60 shown in FIG. 4 represents the effect of handling media data streams including compressed video. The computing spectrum 66 is skewed down to properly align the computing bandwidth requirements with the telecommunications spectrum 62 and the media spectrum 64. Accordingly, current processor technology is not sufficient to handle the transmission and processing associated with complex streams of multi-media data.

The current specialized processor approach to media processing is illustrated in the block diagram shown in FIG. 5. As shown in FIG. 5, special purpose processors are coupled to a back plane 70, which is capable of transmitting instructions and data at the upper kilobits to lower gigabits per second range. In a typical configuration, an audio processor 76, video processor 78, graphics processor 80 and network processor 82 are all coupled to the back plane 70. Each of the audio, video, graphics and network processors 76-82 typically employ their own private or dedicated memories 84, which are only accessible to the specific processor and not accessible over the back plane 70. As described above, however, unless video data streams are constantly being processed, for example, the video processor 78 will sit idle for periods of time. The computing power of the dedicated video processor 78 is thus only available to handle video data streams and is not available to handle other media data streams that are directed to other dedicated processors. This, of course, is an inefficient use of the video processor 78 particularly in view of the overall processing capability of this multi-processor system.

The general purpose media processor 12, in contrast, handles a data stream of audio, video, graphics and network information all at the same time with the same processor. In order to handle the ever changing combination of data types, the general purpose media processor 12 is dynamically partitionable to allocate the appropriate amount of processing for each combination of media in a unified media data stream. A block diagram of two preferred general purpose media processor system configurations is shown in FIG. 6. Referring to FIG. 6, a general purpose media processor 12 is coupled to a high-speed back plane 90. The presently preferred back plane 90 is capable of operation at 30 gigabits per second. As those skilled in the art will appreciate, back planes 90 that are capable of operation at 400 gigabits per second or greater bandwidth are envisioned within the spirit and scope of the invention. Multiple memory devices 92 are also coupled to the back plane 90, which are accessible by the general purpose media processor 12. Input/output devices 94 are coupled to the back plane 90 through a dual-ported memory 92. The configuration of the input/output devices 94 on one end of the dual-ported memory 92 allows the sharing of these memory devices 92 throughout a network 38 of general purpose media processors 12.

Alternatively, FIG. 6 shows a presently preferred integrated general purpose media processor 12. The integrated processor includes on-board memory and I/O 86. The on-board memory is preferably of sufficient size to optimize throughput, and can comprise a cache and/or buffer memory or the like. The integrated media processor 12 also connects to external memory 88, which is preferably larger than the on-board memory 86 and forms the system main memory.

Execution Unit

One presently preferred embodiment of an integrated general purpose media processor 12 is shown in FIG. 7. The

12

core of the integrated general purpose media processor 12 comprises an execution unit 100. Three main elements or subsections are included in the execution unit 100. A multiple precision arithmetic/logic unit ("ALU") 102 performs all logical and simple arithmetic operations on incoming media data streams. Such operations consist of calculate and control operations such as Boolean functions, as well as addition, subtraction, multiplication and division. These operations are performed on single or unified media data streams transmitted to and from the multiple precision ALU 102 over a data bus or data path 108. Preferably the data path 108 is 128 bits wide, although those skilled in the art will appreciate that the data path 108 can take on any width or size without departing from the spirit and scope of the invention. The wider the data path 108 the more unified media data can be processed in parallel by the general purpose media processor 12.

Coupled to the multi-precision ALU 102 via the data path 108, and also an element of the execution unit 100, is a programmable switch 104. The programmable switch 104 performs data handling operations on single or unified media data streams transmitted over the data path 108. Examples of such data handling operations include deals, shuffles, shifts, expands, compresses, swizzles, permutes and reverses, although other data handling operations are contemplated. These operations can be performed on single bits or bit fields consisting of two or more bits up to the entire width of the data path 108. Thus, single bits or bit fields of various sizes can be manipulated through programmable operation of the switch 104.

Examples of the presently preferred data manipulation operations performed by the general purpose media processor 12 are shown in FIG. 8. A group expand operation is visually illustrated in FIG. 8(a). According to the group expand operation, a sequential field of bits 270 can be divided into constituent sub-fields 272a-272d for insertion into a larger field array 274. The reverse of the group expand operation is a group compress or extract operation. A visual illustration of the group compress or extract operation is shown in FIG. 8(b). As shown, separate sub-fields 272a-272d from a larger bit field 274 can be combined to form a contiguous or sequential field of bits 270.

Referring to FIGS. 8(c)-8(e), group deal, shuffle, swizzle and permute operations performed by the programmable switch 104 are also illustrated. The operations performed by these instructions are readily understood from a review of the drawings. The group manipulation operations illustrated in FIGS. 8(a)-8(e) comprise the presently contemplated data manipulation operations for the general purpose media processor 12. As those skilled in the art will appreciate, either a subset of these operations or additional data manipulation operations can be incorporated in other alternate embodiments of the general purpose media processor 12 without departing from the spirit and scope of the invention.

Referring again to FIG. 7, higher level mathematical operations than those performed by the multi-precision ALU 102 are performed in the general purpose media processor 12 through an extended math element 106. The extended math element 106 is coupled to the data path 108 and also comprises part of the execution unit 100. The extended math element 106 performs the complex arithmetic operations necessary for video data compression and similarly intensive mathematical operations. One presently preferred example of an extended math operation comprises a Galois field operation. Other examples of extended mathematical functions performed by the extended math element 106 include CRC generation and checking, Reed-Solomon code genera-

5,794.061

13

tion and checking, and spread-spectrum encoding and decoding. As those skilled in the art appreciate, additional mathematical operations are possible and contemplated.

According to the preferred embodiment of the integrated general purpose media processor 12, a register file 110 is provided in addition to the execution unit 100 to process media data. The register file 110 stores and transmits data streams to and from the execution unit 100 via the data path 108. Rather than employing a complex set of specific or dedicated registers, the general purpose media processor 12 preferably includes 64 general purpose registers in the register file 110 along with one program counter (not shown). The 64 general purpose registers contained in the register file 110 are all available to the user/programmer, and

14

appreciate that a great number of programs are possible through various sequences of instructions. Particular programs can be developed for each unique implementation of the general purpose media processor 12. A detailed discussion of such specific programs is therefore beyond the scope of this description.

One presently preferred instruction set for the general purpose media processor 12 is included in the Microfiche Appendix, the contents of which are hereby incorporated herein by reference. A list of the presently preferred major operation codes for the general purpose media processor 12 appears below in Table I.

TABLE I

MAJOR OPERATION CODES							
MAJOR 0	32	64	96	128	160	192	224
0 ERES	GSHUFFLE	FMULADD16	GMULADD1	LU16LAI	SAAS64LAI	EADDIO	BFE16
1 ESHUFFLE-14MUX	GSHUFFLE-14MUX	FMULADD32	GMULADD2	LU16BAI	SAAS64BAI	EADDIUO	BFNUE16
2	GSELECT8	FMULADD64	GMULADD4	LU16LI	SCAS64LAI	ESETIL	BFNUE16
3 EMDEPI	GMDEPI		GMULADD8	LU16BI	SCAS64BAI	ESETIGE	BFNUL16
4 EMUX	GMUX	FMULSUB16	GMULADD16	LU32LAI	SMAS64LAI	ESETIE	BFE32
5 EBMUX	G8MUX	FMULSUB32	GMULADD32	LU32BAI	SMAS64BAI	ESETINE	BFNUE32
6 ECFMUI64	GCFMUL8	FMULSUB64	GMULADD64	LU32LI	SMUX64LAI	ESETIUL	RFNUE32
7 ETRANPOSE-1MUX	GTRANPOSE-8MUX		GEXTRACT128	LU32BI	SMUX64BAI	ESETIUGE	BFNUL32
8				L16LAI	S16LAI	ESUBIO	BFE64
9 ESWIZZLE	GSWIZZLE		GUMULADD2	L16BAI	S16BAI	ESUBIUO	BFNUE64
10	GSWIZZLECOPY		GUMULADD4	L16LI	S16LI	ESUBIL	BFNUE64
11	GSWIZZLESWAP		GUMULADD8	L16BI	S16BI	ESUBIGE	BFNUL64
12 EDEPI	GDEPI	F.16	GUMULADD16	L32LAI	S32LAI	ESUBIE	BFE128
13 EUDEPI	GUDEPI	F.32	GUMULADD32	L32BAI	S32BAI	ESUBINE	BFNUE128
14 EWTHI	GWTHI	F.64	GUMULADD64	L32LI	S32LI	ESUBIUL	BFNUE128
15 EUWTHI	GUWTHI		GUEXTRACT128	L32BI	S32BI	ESUBIUGE	BFNUL128
16		GFMULADD16	GEXTRACTI	L64LAI	S64LAI	EADDI	BAND
17		GFMULADD32	GEXTRACTI16	L64BAI	S64BAI	EXORI	BANDNE
18		GFMULADD64	GEXTRACTI32	L64LI	S64LI	EORI	BL/BLZ
19		GFMULADD128	GUEXTRACTI64	L64BI	S64BI	EANDI	BGE/BGEZ
20		GFMULSUB16	GEXTRACT	L128LAI	S128LAI	ESUBI	BE
21		GFMULSUB32	I.64	L128BAI	S128BAI		BNE
22		GFMULSUB64	GEXTRACT	L128LI	S128LI	ENORI	BUL/BGZ
23		GFMULSUB128	L.128	L128BI	S128BI	ENANDI	BUCE/BLZ
24			G.1	LBI	SBI		BGATEI
25			G.2	LUBI			
26			G.4				
27			G.8				
28	ECOPYI	GF.16	G.16			ECOPYI	BI
29		GF.32	G.32				BLINKI
30		GF.64	G.64				
31	EMINOR	GF.128	G.128	LMINOR	SMINOR	EMINOR	BMINOR

major operation code field values

50

comprise a portion of the user state of the general purpose media processor 12. The general purpose registers are preferably capable of storing any form of data. Each register within the register file 110 is coupled to the data path 108 and is accessible to the execution unit 100 in the same manner. Thus, the user can employ a general purpose register according to the specific needs of a particular program or unique application. As those skilled in the art will appreciate, the register file 110 can also comprise a plurality of register files 110 configured in parallel in order to support parallel multi-threaded processing.

Instruction Set and User Programming

Control or manipulation of data processed by the general purpose media processor 12 is achieved by selected instructions programmed by the user. Those skilled in the art will

As shown in Table I, the major operation codes are grouped according to the function performed by the operations. The operations are thus arranged and listed above according to the presently preferred operation code number for each instruction. As many as 255 separate operations are contemplated for the preferred embodiment of the general purpose media processor 12. As shown in Table I, however, not all of the operation codes are presently implemented. As those skilled in the art will appreciate, alternate schemes for organizing the operation codes, as well as additional operation codes for the general purpose media processor 12, are possible.

The instructions provided in the instruction set for the general purpose media processor 12 control the transfer, processing and manipulation of data streams between the register file 110 and the execution unit 100. The presently preferred width of the instruction path 112 is 32-bits wide.

5.794.061

15

organized as four eight-bit bytes ("quadlets"). Those skilled in the art will appreciate, however, that the instruction path 112 can take on any width without departing from the spirit and scope of the invention. Preferably, each instruction within the instruction set is stored or organized in memory on four-byte boundaries. The presently preferred format for instructions is shown in FIG. 9(a).

As shown in FIG. 9(a), each of the presently preferred instruction formats for the general purpose media processor 12 includes a field 280 for the major operation code number shown in Table I. Based on the type of operation performed, the remaining bits can provide additional operands according to the type of addressing employed with the operation. For example, the remainder of the 32-bit instruction field can comprise an immediate operand ("imm"), or operands stored in any of the general registers ("ra," "rb," "rc," and "rd"). In addition, minor operation codes 282 can also be included among the operands of certain 32-bit instruction formats.

The presently preferred embodiment of the general purpose media processor 12 includes a limited instruction set similar to those seen in Reduced Instruction Set Computer ("RISC") systems. The preferred instruction set for the general purpose media processor 12 shown in Table I includes operations which implement load, store, synchronize, branch and gateway functions. These five groups of operations can be visually represented as two general classes of related operations. The branch and gateway operations perform related functions on media data streams and are thus visually represented as block 114 in FIG. 7. Similarly, the load, store and synchronize operations are grouped together in block 116 and perform similar operations on the media data streams. (Blocks 114 and 116 only represent the above classification of these operations and their function in the processing of media data streams, and do not indicate any specific underlying electronic connections.) A more detailed discussion of these operations, and the functionality of the general purpose media processor 12, appears in the Microfiche Appendix.

The four-byte structure of instructions for the general purpose media processor 12 is preferably independent of the byte ordering used for any data structures. Nevertheless, the gateway instructions are specifically defined as 16-byte structures containing a code address used to securely invoke a procedure at a higher privilege level. Gateways are preferably marked by protection information specified in the translation lookaside buffer 148 in the memory management unit 122. Gateways are thus preferably aligned on 16-byte boundaries in the external memory. In addition to the general purpose registers and program counter, a privilege level register is provided within the register file 110 that contains the privilege level of the currently executing instruction.

The instruction set preferably includes load and store instructions that move data between memory and the register file 110, branch instructions to compare the content of registers and transfer control, and arithmetic operations to perform computations on the contents of registers. Swap instructions provide multi-thread and multi-processor synchronization. These operations are preferably indivisible and include such instructions as add-and-swap, compare-and-swap, and multiplex-and-swap instructions. The fixed-point compare-and-branch instructions within the instruction set shown in Table I provide the necessary arithmetic tests for equality and inequality of signed and unsigned fixed-point values. The branch through gateway instruction provides a secure means to access code at a higher privileged level in a form similar to a high level language procedure call generally known in the art.

16

The general purpose media processor 12 also preferably supports floating-point compare-and-branch instructions. The arithmetic operations, which are supported in hardware, include floating-point addition, subtraction, multiplication, division and square root. The general purpose media processor 12 preferably supports other floating-point operations defined by the ANSI-IEEE floating-point standard through the use of software libraries. A floating point value can preferably be 16, 32, 64 or 128-bits wide. Examples of the presently preferred floating-point data sizes are illustrated in FIG. 9(b).

The general purpose media processor 12 preferably supports virtual memory addressing and virtual machine operation through a memory management unit 122. Referring to FIG. 10(a), one presently preferred embodiment of the memory management unit 122 is shown. The memory management unit 122 preferably translates global virtual addresses into physical addresses by software programmable routines augmented by a hardware translation lookaside buffer ("TLB") 148. A facility for local virtual address translation 164 is also preferably provided. As those skilled in the art will appreciate, the memory management unit 122 includes a data cache 166 and a tag cache 168 that store data and tags associated with memory sections for each entry in the TLB 148.

A block diagram of one preferred embodiment of the TLB 148 is shown in FIG. 10(b). The TLB 148 receives a virtual address 230 as its input. For each entry in the TLB 148, the virtual address 230 is logically AND-ed with a mask 232. The output of each respective AND gate 234 is compared via a comparator 236 with each entry in the TLB 148. If a match is detected, an output from the comparator 236 is used to gate data 240 through a transceiver 238. As those skilled in the art will appreciate, a match indicates the entry of the corresponding physical address within the contents of the TLB 148 and no external memory or I/O access is required. The data 240 for the data cache 166 (FIG. 10(a)) is then combined with the remaining lower bits of the virtual address 230 through an exclusive-OR gate 242. The resultant combination is the physical address 244 output from the TLB 148. If a match is not detected between the logical address and the contents of the tag cache 168, the memory management unit 122 an external memory or I/O access is necessary to retrieve the relevant portion of memory and update the contents of the TLB 148 accordingly.

Using generally known memory management techniques, the memory management unit 122 ensures that instructions (and data) are properly retrieved from external memory (or other sources) over an external input/output bus 126 (see FIG. 7). As described in more detail below, a high bandwidth interface 124 is coupled to the external input/output bus 126 to communicate instructions (and media data streams) to the general purpose media processor 12. The presently preferred physical address width for the general purpose media processor 12 is eight bytes (64-bits). In addition, the memory management unit 122 preferably provides match bits (not shown) that allow large memory regions to be assigned a single TLB entry allowing for fine grain memory management of large memory sections. The memory management unit 122 also preferably includes a priority bit (not shown) that allows for preferential queuing of memory areas according to respective levels of priority. Other memory management operations generally known in the art are also performed by the memory management unit 122.

Referring again to FIG. 7, instructions received by the general purpose media processor 12 are stored in a combined instruction buffer/cache 118. The instruction buffer/

5,794,061

17

cache 118 is dynamically subdivided to store the largest sequence of instructions capable of execution by the execution unit 100 without the necessity of accessing external memory. In a preferred embodiment of the invention, instruction buffer space is allocated to the smallest and most frequently executed blocks of media instructions. The instruction buffer thus helps maintain the high bandwidth capacity of the general purpose media processor 12 by sustaining the number of instructions executed per second at or near peak operation. That portion of the instruction buffer/cache 118 not used as a buffer is, therefore, available to be used as cache memory. The instruction buffer/cache 118 is coupled to the instruction path 112 and is preferably 32 kilobytes in size.

A data buffer/cache 120 is also provided to store data transmitted and received to and from the execution unit 100 and register file 110. The data buffer/cache 120 is also dynamically subdivided in a manner similar to that of the instruction buffer/cache 118. The buffer portion of the data buffer/cache 120 is optimized to store a set size of unified media data capable of execution without the necessity of accessing external memory. In a preferred embodiment of the invention, data buffer space is allocated to the smallest and most frequently accessed working sets of media data. Like the instruction buffer, the data buffer thus maintains peak bandwidth of the general purpose media processor 12. The data buffer/cache 120 is coupled to the data path 108 and is preferably also 32 kilobytes in size.

The preferred embodiment of the general purpose media processor 12 includes a pipelined instruction pre-fetch structure. Although pipelined operation is supported, the general purpose media processor 12 also allows for non-pipelined operations to execute without any operational penalty. One preferred pipeline structure for the general purpose media processor 12 comprises a "super-string" pipeline shown in FIG. 11. A super-string pipeline is designed to fetch and execute several instructions in each clock cycle. The instructions available for the general purpose media processor 12 can be broken down into five basic steps of operation. These steps include a register-to-register address calculation, a memory load, a register-to-register data calculation, a memory store and a branch operation. According to the super-string pipeline organization of the general purpose media processor 12, one instruction from each of these five types may be issued in each clock cycle. The presently preferred ordering of these operations are as listed above where each of the five steps are assigned letters "A," "L," "E," "S" and "B" (see FIG. 11).

According to the super-string pipelining technique, each of the instructions are serially dependent, as shown in FIG. 11, and the general purpose media processor 12 has the ability to issue a string of dependent instructions in a single clock cycle. These instructions shown in FIG. 11 can take from two to five cycles of latency to execute, and a branch prediction mechanism is preferably used to keep up the pipeline filled (described below). Instructions can be encoded in unit categories such as address, load, store/sync, fixed, float and branch to allow for easy decoding. A similar scheme is employed to pre-fetch data for the general purpose media processor 12.

As those skilled in the art will appreciate, the super-string pipeline can be implemented in a multi-threaded environment. In such an implementation, the number of threads is preferably relatively prime with respect to functional unit rates so that functional units can be scheduled in a non-interfering fashion between each thread.

In another more preferred embodiment, a "super-spring" pipelining scheme is employed with the general purpose

18

media processor 12. The super-spring pipeline technique breaks the super-string pipeline shown in FIG. 11 into two sections that are coupled via a memory buffer (not shown). A visual representation of the super-spring pipeline technique is shown in FIG. 12. The front of the pipeline 204, in which address calculation (A), memory load (L), and branch (B) operations are handled, is decoupled from the back of the pipeline 206, in which data calculation (E) and memory store (S) operations are handled. The decoupling is accomplished through the memory buffer (not shown), which is preferably organized in a first-in-first-out ("FIFO") fast/dense structure. (The memory buffer is functionally represented as a spring in FIG. 12.)

As indicated in Table I above, the general purpose media processor 12 does not include delayed branch instructions, and so relies upon branch or fetch prediction techniques to keep the pipeline full in program flows around unconditional and conditional branch instructions. Many such techniques are generally known in the art. Examples of some presently preferred techniques include the use of group compare and set, and multiplex operations to eliminate unpredictable branches; the use of short forward branches, which cause pipeline neutralization; and where branch and link predicts the return address in a one or more entry stack. In addition, the specialized gateway instructions included in the general purpose media processor 12 allow for branches to and from protected virtual memory space. The gateway instructions, therefore, allow an efficient means to transfer between various levels of privilege.

As described above, two basic forms of media data are processed by the general purpose media processor 12, as shown in FIG. 7. These data streams generally comprise Nyquist sampled I/O 128, and standard memory and I/O 130. As shown in FIG. 7, audio 132, video 134, radio 136, network 138, tape 140 and disc 142 data streams comprise some examples of digitally sampled I/O 128. As those skilled in the art will appreciate, other forms of digitally sampled I/O are contemplated for processing by the general purpose media processor 12 without departing from the spirit and scope of the invention. Standard memory and I/O 130 comprises data received and transmitted to and from general digital peripheral devices used in the design of most computer systems. As shown in FIG. 7, some examples of such devices include dynamic random access memory ("DRAM") 146, or any data received over the PCI bus 144 generally known in the art. Other forms of standard memory and I/O sources are also contemplated. The various fixed-point data sizes preferred for the general purpose media processor 12 are illustrated in FIG. 9(c).

External Interface

As mentioned above, the general purpose media processor 12 includes a high bandwidth interface 124 to communicate with external memory and input/output sources. As part of the high bandwidth interface 124, the general purpose media processor 12 integrates several fast communication channels 156 (FIG. 13) to communicate externally. These fast communication channels 156 preferably couple to external caches 150, which serve as a buffer to memory interfaces 152 coupled to standard memory 154. The caches 150 preferably comprise synchronous static random access memory ("SRAM"), each of which are sixty-four kilobytes in size; and the standard memories 154 comprise DRAM's. The memory interfaces 152 transmit data between the caches 150 and the standard memories 154. The standard memories 154 together form the main external memory for the general purpose media processor 12. The cache 150,

5,794,061

19

memory interface 152, standard memory 154 and input/output channel 156 therefore make up a single external memory unit 158 for the general purpose media processor 12.

According to the presently preferred embodiment of the invention, the memory interface protocol embeds read and write operations to a single memory space into packets containing command, address, data and acknowledgment information. The packets preferably include check codes that will detect single-bit transmission errors and some multiple-bit errors. As many as eight operations may be in progress at a time in each external memory unit 158. As shown in FIG. 13, up to four external memory units 158 may be cascaded together to expand the memory available to the general purpose media processor 12, and to improve the bandwidth of the external memory. Through such cascaded memory units 158, the memory interface 152 provides for the direct connection of multiple banks of standard memory 154 to maintain operation of the general purpose media processor 12 at sustained peak bandwidths.

According to one embodiment shown in FIG. 13, up to four standard memory devices 154 can be coupled to each memory interface 152. Each standard memory 154 thus includes as many as four banks of DRAM, each of which is preferably sixteen bits wide. The standard memories 154 are connected in parallel to the memory interface 152 forming a 72-bit wide data bus 160, where 64 bits are preferably provided for data transfer and eight bits are provided for error correction. In addition to the data bus 160, an address/control bus 162 is coupled between the memory interface 152 and each standard memory 154. The address/control bus 162 preferably comprises at least twelve address lines (4 kilobits \times 16 memory size) and four control lines as shown in FIG. 13. An alternate manner for coupling the DRAM's to the memory interface 152 is illustrated in FIG. 14. As shown in FIG. 14, two banks of four DRAM single inline memory modules are coupled in parallel to the memory interface 152. The memory interface 152 also supports interleaving to enhance bandwidth, and page mode accesses to improve latency for localized addressing.

Using standard DRAM components, the external memory units 158 achieve bandwidths of approximately two gigabits/second with the standard memories 154. When four such external memory units 158 are coupled via the communication channel 156, therefore, the total bandwidth of the external main memory system increases to one gigabyte/second. As discussed further below, in implementations with two or eight communication channels 156, the aggregate bandwidth increases to two and eight gigabytes/second, respectively.

A more detailed depiction of the communication channel 156 circuitry appears in FIG. 15. According to the preferred embodiment of the invention, each communication channel 156 comprises two unidirectional, byte-wide, differential, packet-oriented data channels 156a, 156b (see FIG. 13). As explained above, where memory units 158 are cascaded together in series, the output of one memory unit 158 is connected to the input of another memory unit 158. The two unidirectional channels are thus connected through the memory units 158 forming a loop structure and make up a single bi-directional memory interface channel.

Referring to FIG. 15, each communication channel 156 is preferably eight bits wide, and each bit is transmitted differentially. For example, output transceiver 170 for bit D_{out} transmits both D_0 and \overline{D}_0 signals over the communication channel 156. Additional transceivers are similarly

20

provided for the remaining bits in the channel 156. (The transceiver 176 for bit D_{out} and associated differential lines 178, 180 are shown in FIG. 15.) A CLK_{out} transceiver 182 is also provided to generate differential clock outputs 184, 186 over the channel 156. To complete the link between memory units 158, input transceivers 188-192 are provided in each memory unit 158 for each of the differential bits and clock signals transmitted over the communication channel 156. These input signals 172, 174, 178, 180, 184, 186 are preferably transmitted through input buffers 194-198 to other parts of the memory unit 158 (described above).

Each memory unit 158 also includes a skew calibrator 200 and phase locked loop ("PLL") 202. The skew calibrator 200 is used to control skew in signals output to the communication channel 156. Preferably, digital skew fields are employed, which include set numbers of delay stages to be inserted in the output path of the communication channel 156. Setting these fields, and the corresponding analog skew fields, permits a fine level of control over the relative skew between output channel signals.

The PLL 202 recovers the clock signal on either side of the communication channel 156 and is thus provided to remove clock jitter. The clock signals 184, 186 preferably comprise a single phase, constant rate clock signal. The clock signals 184, 186 thus contain alternating zero and one values transmitted with the same timing as the data signals 172, 174, 178, 180. The clock signal frequency is, therefore, one-half the byte data rate. The communication channel 156 preferably operates at constant frequency and contains no auxiliary control, handshaking or flow control information.

Each external memory unit 158 preferably defines two functional regions: a memory region, implemented by the cache 150 backed by standard memory 154 (see FIG. 13), and a configuration region, implemented by registers (not shown). Both regions are accessed by separate interfaces; the communication channel 156 is used to access the memory region, and a serial interface (described below) is used to access the configuration region. In the memory region, the caches 150 are preferably write-back (write-in) single-set (direct-map) caches for data originally contained in standard memory 154. All accesses to memory space should maintain consistency between the contents of the cache 150 and the contents of the standard memory 154. The configuration region registers provide the mechanism to detect and adjust skew in the communication channel 156. Software is preferably employed to adaptively adjust the skew in the channel 156 through digital skew fields, as explained above. The serial interface thus is used to configure the external memory units 158, set diagnostic modes and read diagnostic information, and to enable the use of a high-speed tester (not shown).

One presently preferred embodiment of the invention employs two byte-wide packet communication channels 156 (FIG. 16(a)). In order to further increase the bandwidth of the general purpose media processor 12, up to sixteen byte-wide packet communication channels 156 can be employed. Referring to FIG. 16(b), twelve communication channels, comprising eight memory channels 210, a ninth channel for parallel processing 212 (described below), and three input/output ("I/O") channels 214, are shown. Each of the communication channels 210-214 preferably employs the cascade configuration of four channel interface devices 216. (Each channel interface device 216 coupled to the memory channels 210 corresponds to the external memory unit 158 shown in FIG. 13.) Through each of the twelve communication channels shown in FIG. 16(b), the general purpose media processor 12 can request or issue read or

5,794,061

21

write transactions. When not interleaved, the twelve channels provide a single contiguous memory space for each channel interface device 216.

Alternatively, memory accesses may be interleaved in order to provide for continuous access to the external memory system at the maximum bandwidth for the DRAM memories. In an interleaved configuration, at any point in time some memory devices will be engaged in row pre-charge, while others may be driving or receiving data, or receiving row or column addresses. The memory interface 152 (FIG. 13) thus preferably maps between a contiguous address space and each of the separate address spaces made available within each external memory unit 158. For maximum performance, therefore, the memory interface is interleaved so that references to adjacent addresses are handled by different memory devices. Moreover, in the preferred embodiment, additional memory operations may be requested before the corresponding DRAM bank is available. In an interleaved approach, these operations are placed in a queue until they can be processed. According to the preferred embodiment, memory writes have lower priority than memory reads, unless an attempt is made to read an address that is queued for a write operation. As those skilled in the art will appreciate, the depth of the memory write queue is dictated by the specific implementation.

Although up to four external memory units 158 are preferably cascaded to form effectively larger memories, some amount of latency may be introduced by the cascade. Packets of data transmitted over the communication channel 156 are uniquely addressed to a particular channel interface device 216. A packet received at a particular device, which specifies another module address, is automatically passed to the correct channel interface device 216. Unless the module address matches a particular device 216, that packet simply passes from the input to the output of the interface device 216. This mechanism divides the serial interconnection of interface devices 216 into strings, which function as a single larger memory or peripheral, but with possibly longer response latency.

In addition to the memory channels 210, the general purpose media processor 12 provides several communication channels 214 for communication with external input/output devices. Referring to FIG. 16(b), three input/output channels 214 having SRAM buffered memory (see FIG. 13) provide an interface to external standard I/O devices (not shown). Like the eight memory channels 210, the three I/O channels 214 are byte-wide input/output channels intended to operate at rates of at least one gigahertz. The three I/O channels 214 also operate as a packet communication link to synchronous SRAM memory 208 within the channel interface device 216. A controller 226 within the channel interface device 216 completes the interface to the I/O devices.

The three I/O channels 214 preferably function in like manner to the memory channels 210 described above. The interface protocol for the three I/O channels 214 divides read and write operations to a single memory space into packets containing command, address, data and acknowledgment information. The packets also include a check code that will detect single-bit transmission errors and some multiple-bit errors. According to the preferred embodiment of the invention, as many as eight operations may progress in each interface device 216 at a time. As shown in FIG. 16(b), up to four channel interface devices 216 can be cascaded together to expand the bandwidth in the three I/O channels 214. A bit-serial interface (not shown) is also provided to each of the channel interface devices 216 to allow access to configuration, diagnostic and tester information at standard

22

TTL signal levels at a more moderate data rate. (A more detailed description of the serial interface is provided below).

Like the memory channels 210, each I/O channel 214 includes nine signals—one clock signal and eight data signals. Differential voltage levels are preferably employed for each signal. Each channel interface device 216 is preferably terminated in a nominal 50 ohm impedance to ground. This impedance applies for both inputs and outputs to the communication channel 156. A programmable termination impedance is preferred.

Interface Communication

According to one presently preferred embodiment of the invention, the channel interface devices 216 can operate as either master devices or slave devices. A master device is capable of generating a request on the communication channel 156 and receiving responses from the communication channel 156. Slave devices are capable of receiving requests and generating responses, over the communication channel 156. A master device is preferably capable of generating a constant frequency clock signal and accepting signals at the same clock frequency over the communication channel 156. A slave device, therefore, should operate at the same clock rate as the communication channel 156, and generate no more than a specified amount of variation in output clock phase relative to input clock phase. The master device, however, can accept an arbitrary input clock phase and tolerates a specified amount of variation in clock phase over operating conditions.

Packets of information sent over the communication channel 156 preferably contain control commands, such as read or write operations, along with addresses and associated data. Other commands are provided to indicate error conditions and responses to the above commands. When the communication channel 156 is idle, such as during initialization and between transmitted packets, an idle packet, consisting of an all-zero byte and an all-one byte is transmitted through the communication channel 156. Each non-idle packet consists of two bytes or a multiple of two bytes, and begins with a byte having a value other than all zeros. All packets transmitted over the communication channel 156 also begin during a clock period in which the clock signal is zero, and all packets preferably end during a clock period in which the clock signal is one. A depiction of the preferred packet protocol format for transmission over the communication channel 156 appears in FIG. 17.

The general form of each packet is an array of bytes preferably without a specific byte ordering. The first byte contains a module address 250 ("ma") in the high order two bits; a packet identifier, usually a command 252 ("com"), in the next three bit positions; and a link identification number 254 ("lid") in the last three bit positions. The interpretation of the remaining bytes of a packet depend upon the contents of the packet identifier. The length of each packet is preferably implied by the command specified in the initial byte of the packet. A check byte is provided and computed as odd bit-wise parity with a leftward circular rotation after accumulating each byte. This technique provides detection of all single-bit and some multiple-bit errors, but no correction is provided.

The modular address 250 field of each packet is preferably a two-bit field and allows for as many as four slave devices to be operated from a single communication channel 156. Module address values can be assigned in one of two fashions: either dynamically assigned through a configura-

5,794,061

23

tion register (not shown), or assigned via static/geometric configuration pins. Dynamic assignment through a configuration register is the presently preferred method for assigning module address values.

The link identification number 254 field is preferably 3-bits wide and provides the opportunity for master devices to initiate as many as eight independent operations at any one time to each slave device. Each outstanding operation requires a distinct link identification number, but no ordering of operations should be implied by the value of the link identification field. Thus, there is preferably no requirement for link identification values 254 to be sequentially assigned either in requests or responses.

The receipt of packets over the communication channel 156 that do not conform to the channel protocol preferably generates an error condition. As those skilled in the art will appreciate, the level or degrees to which a specific implementation detects errors is defined by the user. In one presently preferred embodiment of the invention, all errors are detected, and the following protocol is employed for handling errors. For each error detected, the channel interface device 216 causes a response explicitly indicating the error condition. Channel interface devices 216 reporting an invalid packet will then suppress the receipt of additional packets until the error is cleared. The transmitted packet is otherwise ignored. However, even though the erroneous packet is ignored, the channel interface devices 216 preferably continue to process valid packets that have already been received and generate responses thereto. An identification of the presently preferred commands 252 to be used over the communication channel 156 are listed in FIG. 17.

In the master/slave preferred embodiment, the channel interface devices 216 forward packets that are intended for other devices connected to the communication channel 156, as described above. In slave devices, forwarding is performed based on the module address 250 field of the packet. Packets which contain a module address 250 other than that of the current device are forwarded on to the next device. All non-idle packets are thus forwarded including error packets. In master devices, forwarding is performed based on the link identifier number 254 of the packet. Packets that contain link identifier numbers 254 not generated by the specific channel interface device 216 are forwarded. In order to reduce transmission latency, a packet buffer may be provided. As those skilled in the art appreciate, the suitable size for the packet buffer depends on the amount of latency tolerable in a particular implementation.

A variety of master/slave ring configurations are possible using the high bandwidth interface 124 of the invention. Five ring configurations are currently preferred: single-master, dual-master, multiple-master, single-slave and multiple-master/multiple-slave. The simplest ring configuration contains a single non-forwarding master device and a single non-forwarding slave device. No forwarding is required for either device in this configuration as packets are sent directly to the recipient. A single-master ring, however, may contain a cascade of up to four slave devices (see FIGS. 13, 16). In the single-master ring configuration, each slave device is configured to a distinct module address, and each slave device forwards packets that contain module address fields unequal to their own. As discussed above, a single-master ring provides a larger memory or I/O capacity than a master-slave pair, but also introduces a potentially longer response latency. In the single-master ring, each slave device may have as many as eight transactions outstanding at any time, as described above.

The remaining combinations share many of the above basic attributes. In a dual-master pair, each master device

24

may initiate read and write operations addressed to the other, and each may have up to eight such transactions outstanding. No forwarding is required for either device because packets are sent directly to the recipient. A multiple-master ring may contain multiple master devices and a single slave device. In this configuration, the slave device need not forward packets as all input packets are designated for the single slave device. A multiple-master ring may contain multiple master devices and as many as four slave devices. Each slave device may have up to eight transactions outstanding, and each master device may use some of those transactions. In a preferred embodiment, a master also has the capability to detect a time-out condition or when a response to a request packet is not received. Further aspects of interprocessor communications and configurations are discussed below in connection with FIG. 18.

Serial Bus

In one preferred embodiment of the invention, the general purpose media processor 12 includes a serial bus (not shown). The serial bus is designed to provide bootstrap resources, configuration, and diagnostic support to the general purpose media processor 12. The serial bus preferably employs two signals, both at TTL levels, for direct communication among many devices. In the preferred embodiment, the first signal is a continuously running clock, and the second signal is an open-collector bi-directional data signal. Four additional signals provide geographic addresses for each device coupled to the serial bus. A gateway protocol, and optional configurable addressing, each provide a means to extend the serial bus to other buses and devices. Although the serial bus is designed for implementation in a system having a general purpose media processor 12, as those skilled in the art will appreciate, the serial bus is applicable to other systems as well.

Because the serial bus is preferably used for the initial bootstrap program load of the general purpose media processor 12, the bootstrap ROM is coupled to the serial bus. As a result, the serial bus needs to be operational for the first instruction fetch. The serial bus protocol is therefore devised so that no transactions are required for initial bus configuration or bus address assignment.

According to the preferred embodiment, the clock signal comprises a continuously running clock signal at a minimum of 20 megahertz. The amount of skew, if any, in the clock signal between any two serial bus devices should be limited to be less than the skew on the data signal. Preferably, the serial data signal is a non-inverted open collector bi-directional data signal. TTL levels are preferred for communication on the serial bus, and several termination networks may be employed for the serial data signal. A simple preferred termination network employs a resistive pull-up of 220 ohms to 3.3 volts above V_{SS} . An alternate embodiment employs a more complex termination network such as a termination network including diodes or the "Forced Perfect Termination" network proposed for the SCSI-2 standard, which may be advantageous for larger configurations.

The geographic addressing employed in the serial bus is provided to insure that each device is addressable with a number that is unique among all devices on the bus and which also preferably reflects the physical location of the device. Thus, the address of each device remains the same each time the system is operated. In one preferred embodiment, the geographic address is composed of four bits, thus allowing for up to 16 devices. In order to extend

5,794,061

25

the geographic addressing to more than 16 devices. additional signals may be employed such as a buffered copy of the clock signal or an inverted copy of the clock signal (or both).

The serial bus preferably incorporates both a bit level and packet protocol. The bit level protocol allows any device to transmit one bit of information on the bus, which is received by all devices on the bus at the same time. Each transmitted bit begins at the rising edge of the clock signal and ends at the next rising edge. The transmitted bit value is sampled at the next rising edge of the clock signal. According to one preferred embodiment where the serial data signal is an open collector signal, the transmission of a zero bit value on the bus is achieved by driving the serial data signal to a logical low value. In this embodiment, the transmission of a one bit value is achieved by releasing the serial data signal to obtain a logical high value. If more than one device attempts to transmit a value on the same clock, the resulting value is a zero if any device transmits a zero value, and one if all devices transmit a one value. This provides a "wired-AND" collision mechanism as those skilled in the art will appreciate. If two or more devices transmit the same value on the same clock cycle, however, no device can detect the occurrence of a collision. In such cases, the transaction, which may occur frequently in some implementations, preferably proceeds as described below.

The packet protocol employed with the serial bus uses the bit level protocol to transmit information in units of eight bits or multiples of eight bits. Each packet transmission preferably begins with a start bit in which the serial data signal has a zero (driven) value. After transmitting the eight data bits, a parity bit is transmitted. The transmission continues with additional data. A single one (released) bit is transmitted immediately following the least significant bit of each byte signaling the end of the byte.

On the cycle following the transmission of the parity bit, any device may demand a delay of two cycles to process the data received. The two cycle delay is initiated by driving the serial data signal (to a zero value) and releasing the serial data signal on the next cycle. Before releasing the serial data signal, however, it is preferable to insure that the signal is not being driven by any other device. Further delays are available by repeating this pattern.

In order to avoid collisions, a device is not permitted to start a transmission over the serial bus unless there are no currently executing transactions. To resolve collisions that may occur if two devices begin transmission on the same cycle, each transmitting device should preferably monitor the bus during the transmission of one (released) bits. If any of the bits of the byte are received as zero when transmitting a one, the device has lost arbitration and must cease transmission of any additional bits of the current byte or transaction.

According to the preferred embodiment of the invention, a serial bus transaction consists of the transmission of a series of packets. The transaction begins with a transmission by the transaction initiator, which specifies the target network, device, length, type and payload of the transaction request. The transaction terminates with a packet having a type field in a specified range. As a result, all devices connected to the serial bus should monitor the serial data signal to determine when transactions begin and end. A serial bus network may have multiple simultaneous transactions occurring, however, so long as the target and initiator network addresses are all disjoint.

Parallel Processing

In one preferred embodiment of the invention, two or more general purpose media processors 12 can be linked

26

together to achieve a multiple processor system. According to this embodiment, general purpose media processors 12 are linked together using their high bandwidth interface channels 124, either directly or through external switching components (not shown). The dual-master pair configuration described above can thus be extended for use in multiple-master ring configurations. Preferably, internal daemons provide for the generation of memory references to remote processors, accesses to local physical memory space, and the transport of remote references to other remote processors. In a multi-processor environment, all general purpose media processors 12 run off of a common clock frequency, as required by the communication channels 156 that connect between processors.

Referring to FIG. 18, each general purpose media processor 12 preferably includes at least a pair of inter-processor links 218 (see also FIG. 16(b)). In one configuration, both pairs of inter-processor links 218 can be connected between the two processors 12 to further enhance bandwidth. As shown in FIG. 18(a) several processors 12 may be interconnected in a linear network employing the transponder daemons in each processor. In an alternate embodiment shown in FIG. 18(b), the inter-processor links 218 may be used to join the general purpose media processors 12 in a ring configuration. Alternatively still, general purpose media processors 12 may be interconnected into a two-dimensional network of processors of arbitrary size, as shown in FIG. 18(c). Sixteen processors are connected in FIG. 18(c) by connecting four ring networks. In yet another alternate embodiment, by connecting the inter-processor links 218 to external switching devices (not shown), multi-processors with a large number of processors can be constructed with an arbitrary interconnection topology.

The requester, responder and transponder daemons preferably handle all inter-processor operations. When one general purpose media processor 12 attempts a load or store to a physical address of a remote processor, the requester daemon autonomously attempts to satisfy the remote memory reference by communicating with the external device. The external device may comprise another processor 12 or a switching device (not shown) that eventually reaches another processor 12. Preferably, two requester daemons are provided each processor 12, which act concurrently on two different byte channels and/or module addresses. The responder daemon accepts writes from a specified channel and module address, which enables an external device to generate transaction requests in local memory or to generate processor events. The responder daemon also generates link level writes to the same external device that communicated responses for the received transaction request. Two such responder daemons are preferably provided; each of which operate concurrently to two different byte channels and/or module addresses.

The transponder daemon accepts writes from a specified channel and module address, which enable an external device to cause a requester daemon to generate a request on another channel and module address. Preferably, two such transponder daemons are provided, each of which act concurrently (back-to-back) between two different byte channel and/or module addresses. As those skilled in the art will appreciate, the requester, responder and transponder daemons must act cooperatively to avoid deadlock that may arise due to an imbalance of requests in the system. Deadlocks prevent responses from being routed to their destinations, which may defeat the benefits of a multi-processor distributed system.

According to one presently preferred embodiment of the invention, the general purpose media processor 12 can be

5,794,061

27

implemented as one or more integrated circuit chips. Referring to FIG. 19, the presently preferred embodiment of the general purpose media processor 12 consists of a four-chip set. In the four-chip set, a general purpose media processor 12 is manufactured as a stand alone integrated circuit. The stand alone integrated circuit includes a memory management unit 122, instruction and data cache/buffers 118, 120, and an execution unit 100. A plurality of signal input/output pads 260 are provided around the circumference of the integrated circuit to communicate signals to and from the general purpose media processor 12 in a manner generally known in the art.

The second and third chips of the four-chip set comprise in an external memory element 158 and a channel interface device 216. The external memory element 158 includes an interface to the communication channel 156, a cache 150 and a memory interface 152. The channel interface device 216 also includes an interface to the communication channel 156, as well as buffer memory 262, and input/output interfaces 264. Both the external memory element 158 and the channel interface device 216 include a plurality of input/output signal pads 260 to communicate signals to and from these devices in a generally known manner.

The fourth integrated circuit chip comprises a switch 226, which allows for installation of the general purpose media processor 12 in the heterogeneous network 38. In addition to the plurality of input/output pads 260, the switch 226 includes an interface to the communication channel 156. The switch 226 also preferably includes a buffer 262, a router 266, and a switch interface 268.

As those skilled in the art will appreciate, many implementations for the general purpose media processor 12 are possible in addition to the four-chip implementation described above. Rather than an integrated approach, the general purpose media processor can be implemented in a discrete manner. Alternatively, the general purpose media processor 12 can be implemented in a single integrated circuit, or in an implementation with fewer than four integrated circuit chips. Other combinations and permutations of these implementations are contemplated.

There has been described a system for processing streams of media data at substantially peak rates to allow for real time communication over a large heterogeneous network. The system includes a media processor at its core that is capable of processing such media data streams. The heterogeneous network consists of, for example, the fiber optic/coaxial cable/twisted wire network in place throughout the U.S. To provide for such communication of media data, a media processor according to the invention is disposed at various locations throughout the heterogeneous network. The media processor would thus function both in a server capacity and at an end user site within the network. Examples of such end user sites include televisions, set-top converter boxes, facsimile machines, wireless and cellular telephones, as well as large and small business and industrial applications.

To achieve such high rates of data throughput, the media processor includes an execution unit, high bandwidth interface, memory management unit, and pipelined instruction and data paths. The high bandwidth interface includes a mechanism for transmitting media data streams to and from the media processor at rates at or above the gigahertz frequency range. The media data stream can consist of transmission, presentation and storage type data transmitted alone or in a unified manner. Examples of such data types include audio, video, radio, network and digital communi-

28

cations. According to the invention, the media processor is dynamically partitionable to process any combination or permutation of these data types in any size.

A programmable, general purpose media processor system presents significant advantages over current multimedia communications. Rather than rigid, costly and inefficient specialized processors, the media processor provides a general purpose instruction set to ease programmability in a single device that is capable of performing all of the operations of the specialized processor combination. Providing a uniform instruction set for all media related operations eliminates the need for a programmer to learn several different instruction sets, each for a different specialized processor. The complexity of programming the specialized processors to work together and communicate with one another is also greatly reduced. The unified instruction set is also more efficient. Highly specialized general calculation instructions that are tailored to general or special types of calculations rather than enhancing communication are eliminated.

Moreover, the media processor system can be easily reprogrammed simply by transmitting or downloading new software over the network. In the specialized processor approach, new programming usually requires the delivery and installation of new hardware. Reprogramming the media processor can be done electronically, which of course is quicker and less costly than the replacement of hardware.

It is to be understood that a wide range of changes and modifications to the embodiments described above will be apparent to those skilled in the art and are contemplated. It is therefore intended that the foregoing detailed description be regarded as illustrative rather than limiting, and that it be understood that it is the following claims, including all equivalents, that are intended to define the spirit and scope of this invention.

We claim:

1. A general purpose programmable media processor having an instruction path and a data path to digitally process a plurality of media data streams, comprising:

a high bandwidth external interface operable to receive a plurality of data of various sizes from an external source and communicate the received data over the data path at a rate that maintains substantially peak operation of the media processor;

at least one register file configurable to receive and store data from the data path and to communicate the stored data to the data path; and

a multi-precision execution unit coupled to the data path, the multi-precision execution unit configurable to dynamically partition data received from the data path to account for the elemental symbol width of the plurality of media data streams, said elemental symbol width being equal to or narrower than the data path, and programmable to operate on the data to generate a unified symbol output to the data path.

2. The media processor defined in claim 1, wherein the execution unit is dynamically configurable to partition data received from the data path.

3. The media processor defined in claim 1, further comprising:

means for moving data between registers and memory by performing load and store operations, and for coordinating the sharing of data among a plurality of tasks by performing synchronization operations based upon instructions and data received by the execution unit;

means for securely controlling the sequence of execution by performing branch and gateway operations based upon instructions and data received by the execution unit; and

5.794.061

29

a memory management unit, the memory management unit operable to retrieve data and instructions for timely and secure communication over the data path and instruction path.

4. The media processor defined in claim 3, further comprising:

a combined instruction cache and buffer, the combined instruction cache and buffer dynamically allocated between cache space and buffer space to ensure real-time execution of multiple media instruction streams; and

a combined data cache and buffer, the combined data cache and buffer dynamically allocated between cache space and buffer space to ensure real-time response for multiple media data streams.

5. The media processor defined in claim 4, wherein real-time execution is ensured by dynamically allocating instruction buffer space to the smallest and most frequently executed blocks of media instructions.

6. The media processor defined in claim 4, wherein real-time response is ensured by dynamically allocating data buffer space to the smallest and most frequently accessed working sets of media data.

7. The media processor defined in claim 1, wherein media data streams comprise Nyquist sampled inputs and outputs.

8. The media processor defined in claim 1, wherein media data streams originate from standard computer memory and I/O interfaces.

9. The media processor defined in claim 1, wherein the multi-precision execution unit is configurable to divide the data into component symbols of various sizes, analyze the component symbols based upon instructions, and resynthesize the component symbols for communication over the data path.

10. The media processor defined in claim 1, wherein the plurality of media data streams comprise presentation media information, transmission media information, and storage media information.

11. The media processor defined in claim 10, wherein presentation media information comprises audio, video, image, and graphical information.

12. The media processor defined in claim 10, wherein transmission media information comprises radio and network data transmissions.

13. The media processor defined in claim 10, wherein storage media information comprises data encoded in moving and solid-state memory media.

14. The media processor defined in claim 1, wherein the width of the data path is at least 128 bits.

15. The media processor defined in claim 1, wherein the multi-precision execution unit comprises a dynamically partitionable arithmetic unit, a register controllable cross-bar switch, and an extended mathematical element.

16. The media processor defined in claim 13, wherein the register controllable cross-bar switch comprises a Benes network design.

17. The media processor defined in claim 15, wherein the register controllable cross-bar switch is programmable and is operable to manipulate symbols.

18. The media processor defined in claim 11, wherein the extended mathematical element is operable to perform finite group, finite field, finite ring and table look-up operations on the symbols.

30

19. The media processor defined in claim 1, further comprising a set of predefined instructions accessible by a user.

20. The media processor defined in claim 2, wherein the means for performing load, store, and synchronization operations and the means for performing branch and gateway operations comprises a set of predefined instructions accessible by a user.

21. The media processor defined in claim 20, wherein the predefined instructions are combinable to implement composite functions on the plurality of media data streams.

22. A parallel multi-processor system that maintains substantially peak data throughput in the unified execution of a plurality media data streams, the system having a data path, comprising:

at least one high bandwidth external interface, the at least one high bandwidth external interface coupled to the data path and operable to receive a plurality of data of various sizes from an external source and communicate the received data over the data path at a rate that maintains substantially peak operation of the parallel multi-processor system;

a plurality of register files, each register file having at least one general purpose register coupled to the data path and operable to store a working set of media data received from the data path and to communicate the stored data to the data path; and

at least one multi-precision execution unit coupled to the data path, the at least one multi-precision execution unit configurable to dynamically partition data within the working set of media data received from the data path to account for the elemental symbol width of the plurality of media data streams, said elemental symbol width being equal to or narrower than the data path, and programmable to operate in parallel on the dynamically partitioned data to generate a unified symbol output for each register file.

23. The parallel multi-processor system defined in claim 22, wherein the at least one execution unit alternates in a round robin manner to operate on data stored in the plurality of register files.

24. The parallel multi-processor system defined in claim 22, further comprising an instruction pre-fetch pipeline.

25. The parallel multi-processor system defined in claim 24, wherein the instruction pre-fetch pipeline comprises a super-string pipeline.

26. The parallel multi-processor system defined in claim 24, wherein the instruction pre-fetch pipeline comprises a super-spring pipeline.

27. The parallel multi-processor system defined in claim 22, further comprising a data pre-fetch pipeline.

28. The parallel multi-processor system defined in claim 27, wherein the data pre-fetch pipeline comprises a super-string pipeline.

29. The parallel multi-processor system defined in claim 27, wherein the data pre-fetch pipeline comprises a super-spring pipeline.

30. The parallel multi-processor system defined in claim 22, further comprising a requester, responder and transponder daemon.

* * * * *

EXHIBIT 8



US005809321A

United States Patent [19]
Hansen et al.

[11] **Patent Number:** **5,809,321**
 [45] **Date of Patent:** **Sep. 15, 1998**

[54] **GENERAL PURPOSE, MULTIPLE
 PRECISION PARALLEL OPERATION,
 PROGRAMMABLE MEDIA PROCESSOR**

[75] Inventors: **Craig Hansen**, Los Altos; **John
 Moussouris**, Palo Alto, both of Calif.

[73] Assignee: **MicroUnity Systems Engineering,
 Inc.**, Sunnyvale, Calif.

[21] Appl. No.: **754,429**

[22] Filed: **Nov. 22, 1996**

Related U.S. Application Data

[62] Division of Ser. No. 516,036, Aug. 16, 1995, Pat. No.
 5,742,840.

[51] Int. Cl.⁶ **G06F 9/00**

[52] U.S. Cl. **395/800.01**

[58] Field of Search **395/800.01, 670,
 395/376, 280; 364/131-134, 736, 741,
 745, 748, 754, 761, 768**

[56] References Cited

U.S. PATENT DOCUMENTS

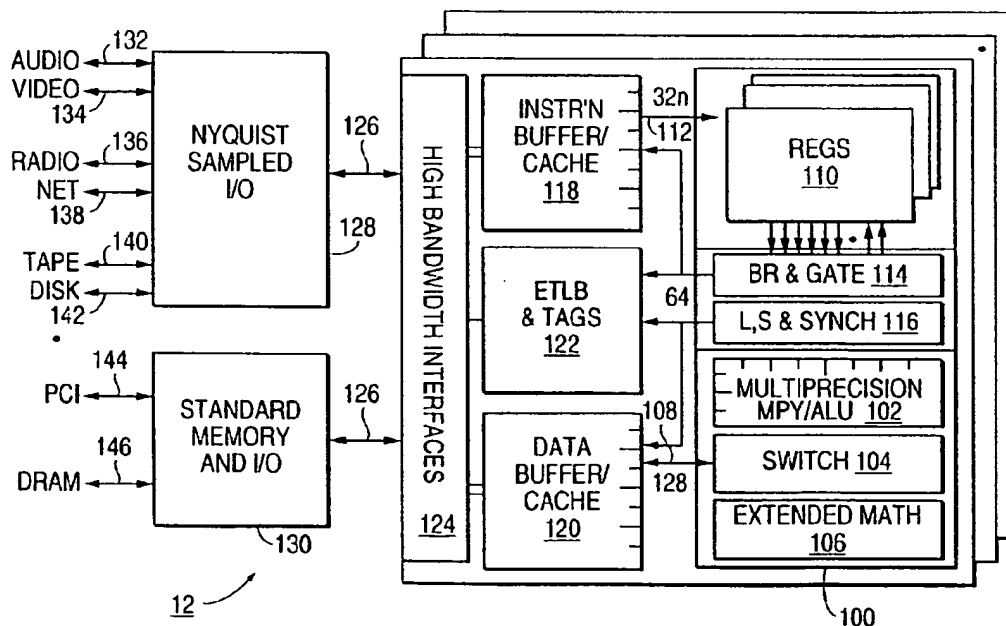
4,893,267	1/1990	Alsup et al.	364/745
4,975,868	12/1990	Freerksen	364/748
5,201,056	4/1993	Daniel et al.	395/800
5,268,855	12/1993	Mason et al.	364/748
5,426,600	6/1995	Nakagawa et al.	364/764

Primary Examiner—Alpesh M. Shah
Attorney, Agent, or Firm—McDermott, Will & Emery

[57] ABSTRACT

A general purpose, programmable media processor for processing and transmitting a media data stream of audio, video, radio, graphics, encryption, authentication, and networking information in real-time. The media processor incorporates an execution unit that maintains substantially peak data throughout of media data streams. The execution unit includes a dynamically partitionable multi-precision arithmetic unit, programmable switch and programmable extended mathematical element. A high bandwidth external interface supplies media data streams at substantially peak rates to a general purpose register file and the multi-precision execution unit. A memory management unit, and instruction and data cache/buffers are also provided. High bandwidth memory controllers are linked in series to provide a memory channel to the general purpose, programmable media processor. The general purpose, programmable media processor is disposed in a network fabric consisting of fiber optic cable, coaxial cable and twisted pair wires to transmit, process and receive single or unified media data streams. Parallel general purpose media processors are disposed throughout the network in a distributed virtual manner to allow for multi-processor operations and sharing of resources through the network. A method for receiving, processing and transmitting media data streams over the communications fabric is also provided.

8 Claims, 25 Drawing Sheets

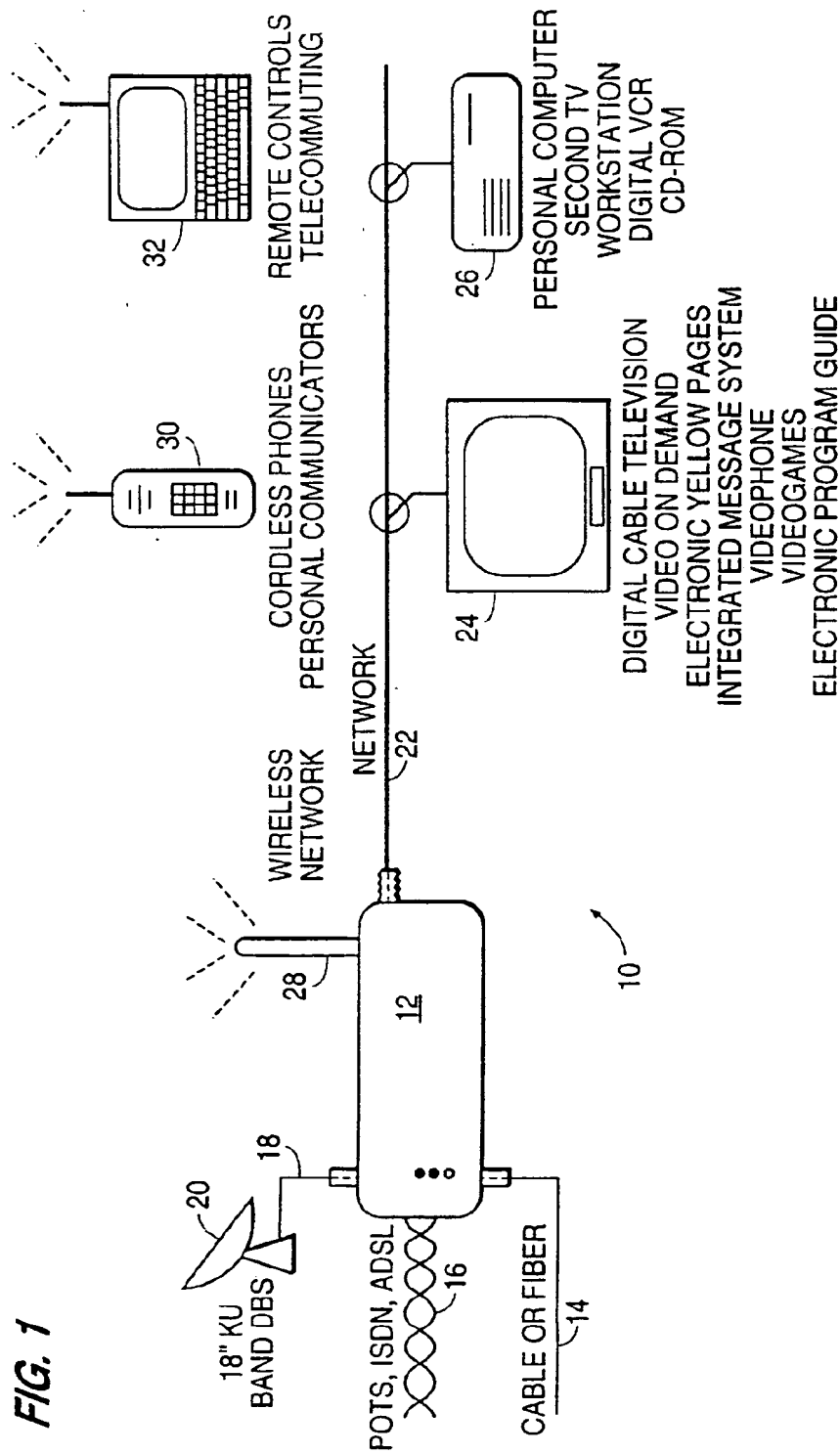


U.S. Patent

Sep. 15, 1998

Sheet 1 of 25

5,809,321



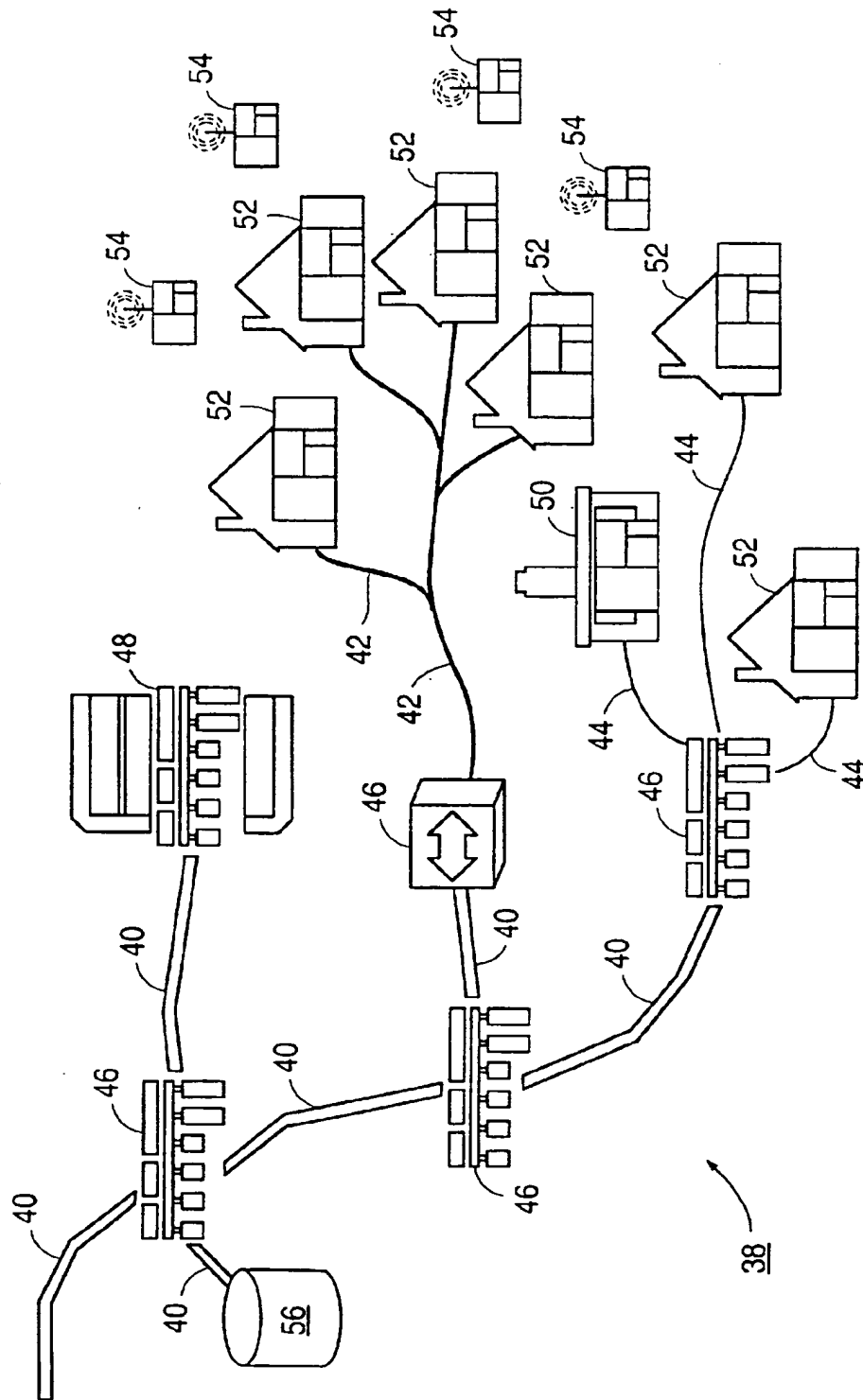
U.S. Patent

Sep. 15, 1998

Sheet 2 of 25

5,809,321

FIG. 2

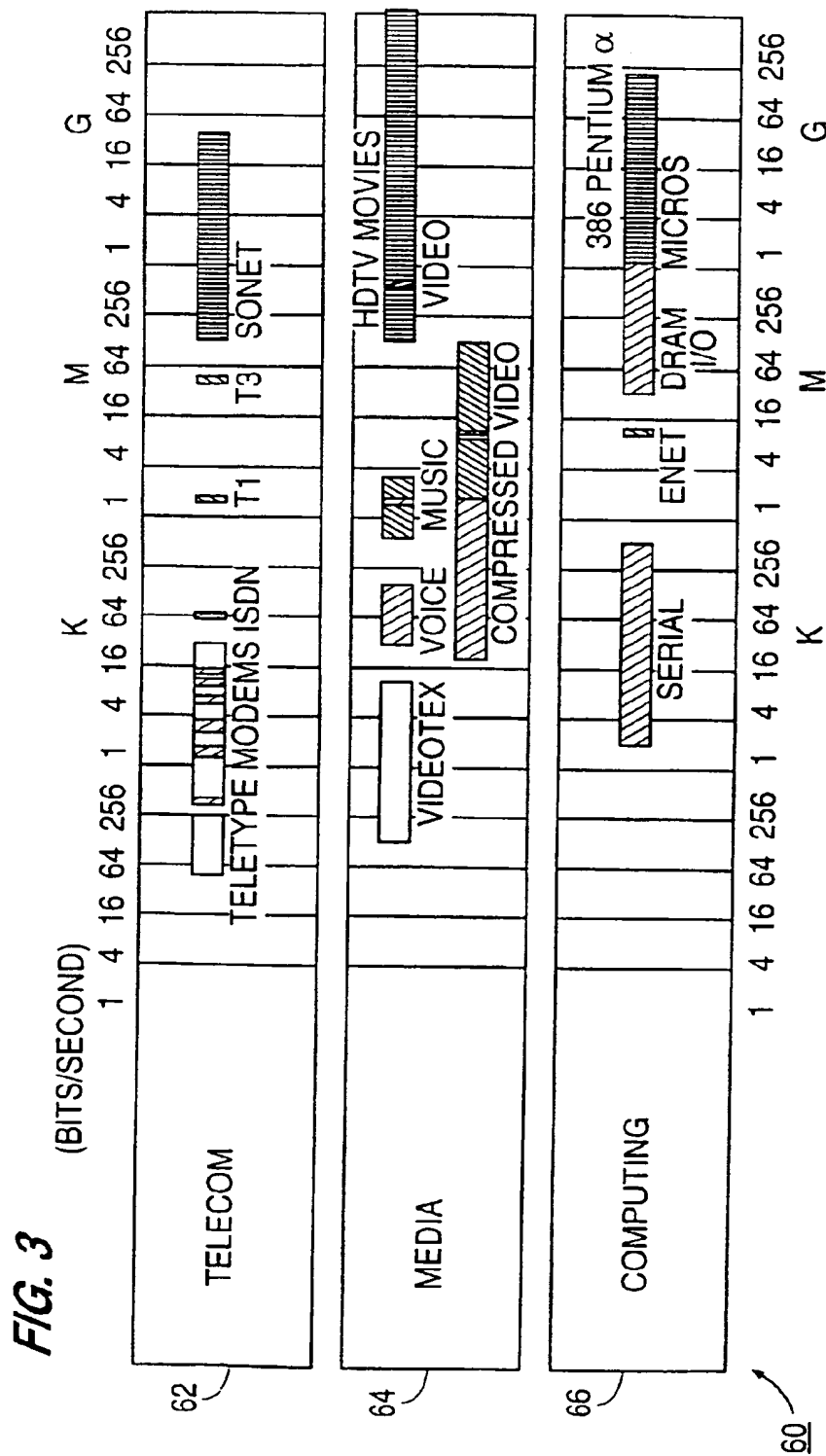


U.S. Patent

Sep. 15, 1998

Sheet 3 of 25

5,809,321

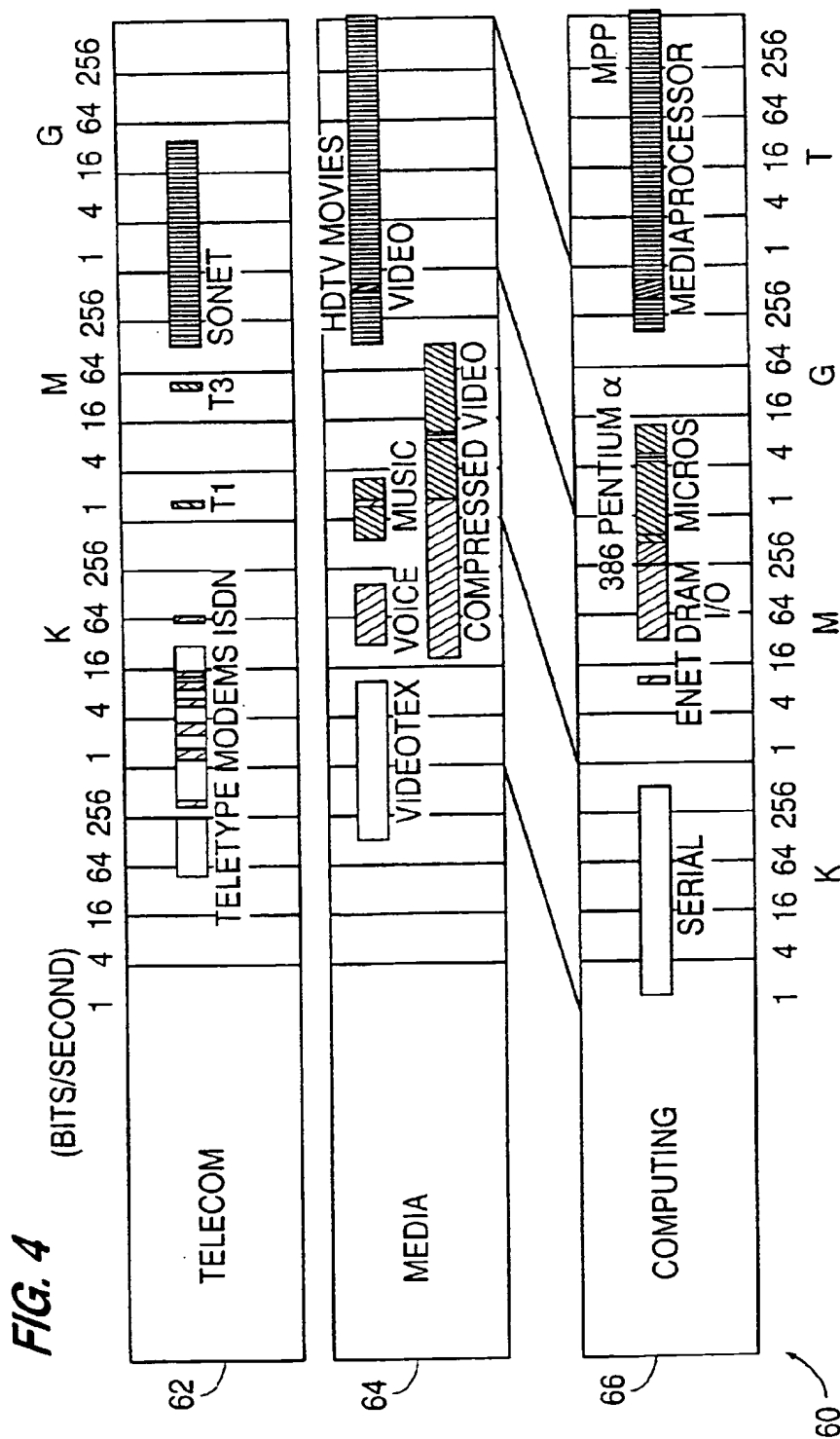


U.S. Patent

Sep. 15, 1998

Sheet 4 of 25

5,809,321



U.S. Patent

Sep. 15, 1998

Sheet 5 of 25

5,809,321

FIG. 5

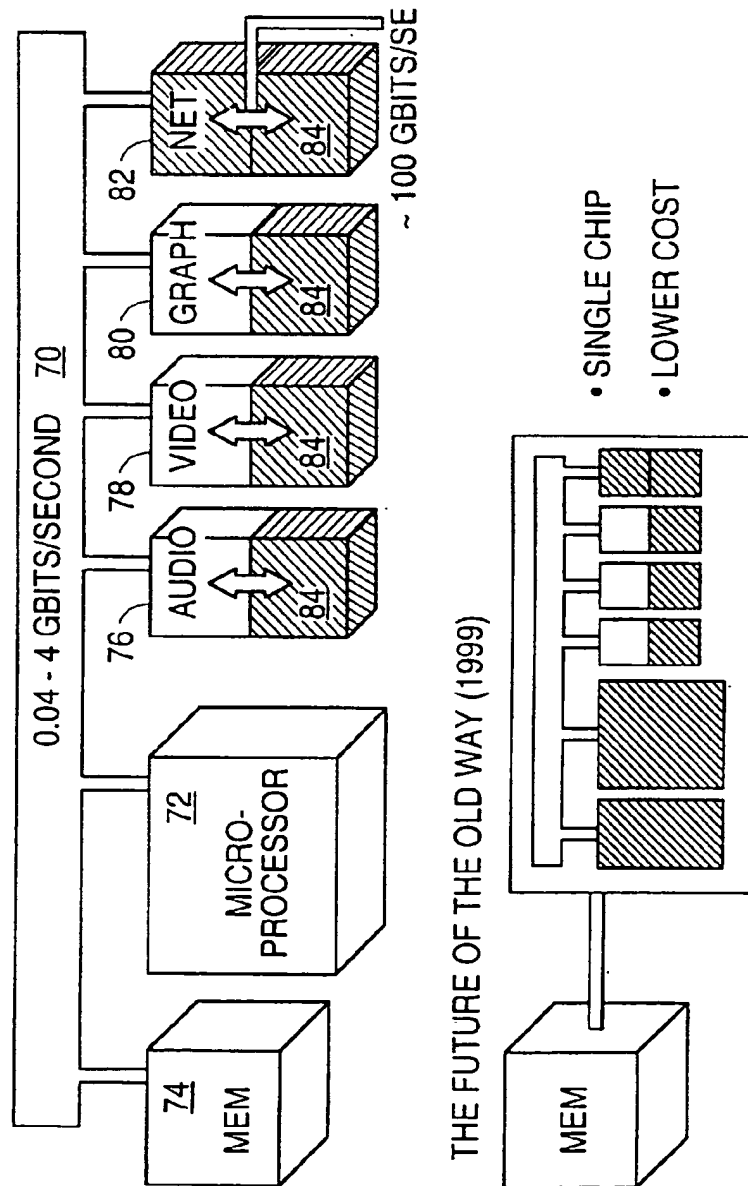
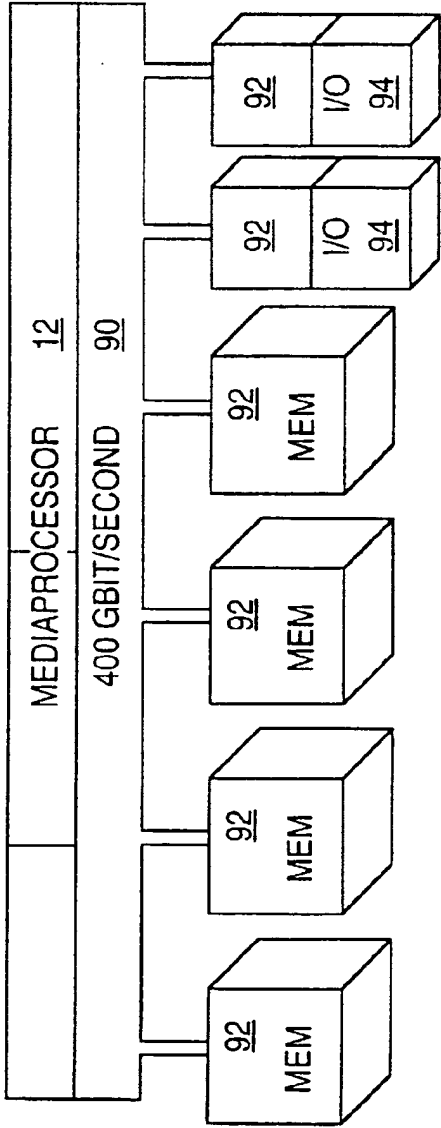
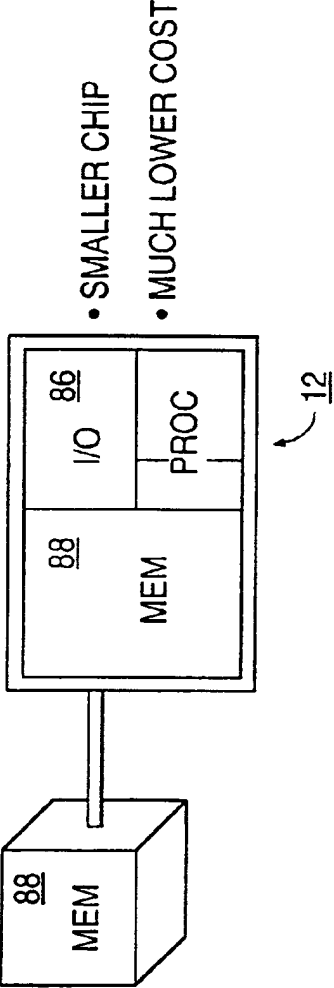


FIG. 6



THE FUTURE OF THE UNIFIED WAY (1995)



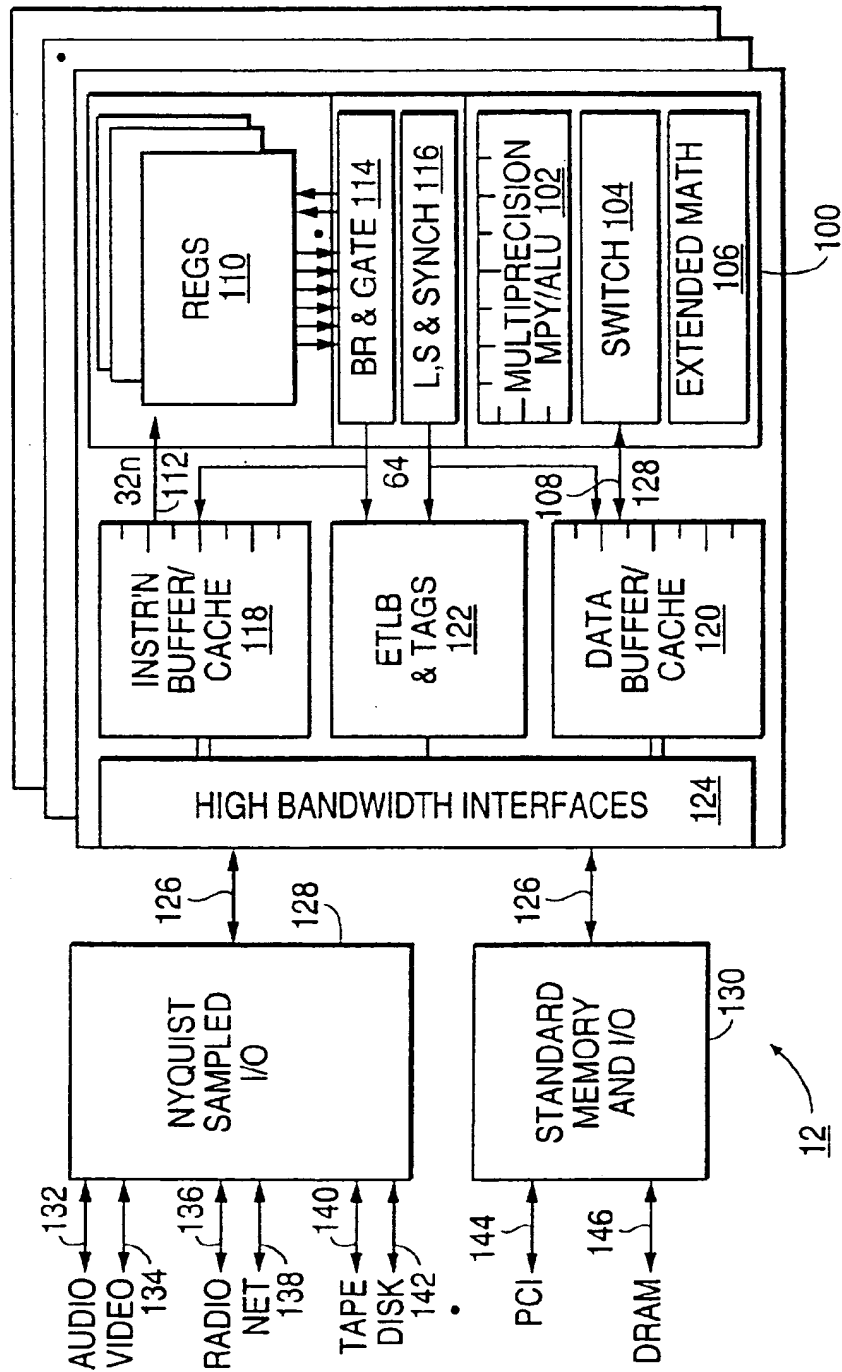
U.S. Patent

Sep. 15, 1998

Sheet 7 of 25

5,809,321

FIG. 7



U.S. Patent

Sep. 15, 1998

Sheet 8 of 25

5,809,321

FIG. 8(a)

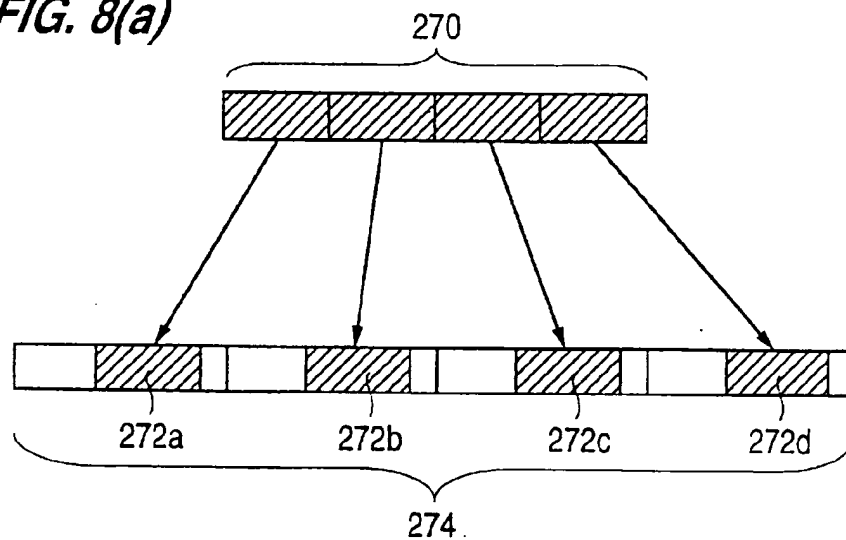
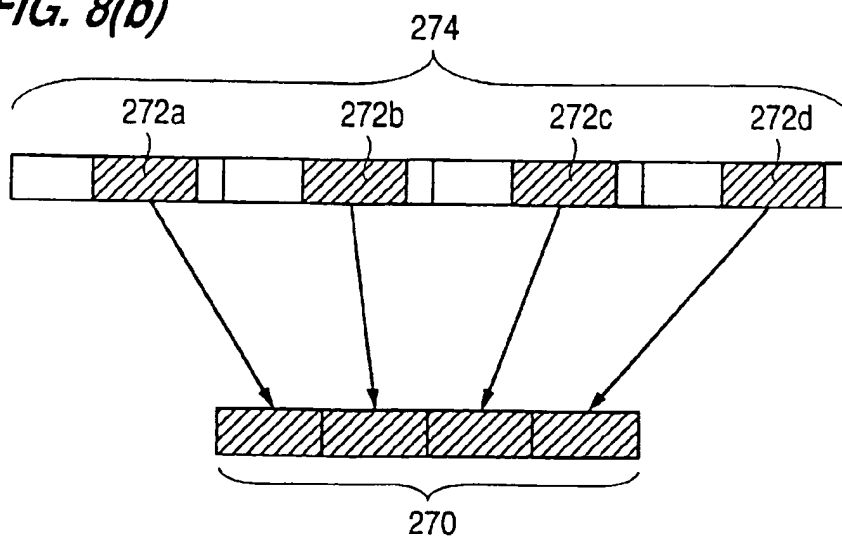


FIG. 8(b)



U.S. Patent

Sep. 15, 1998

Sheet 9 of 25

5,809,321

FIG. 8(c)

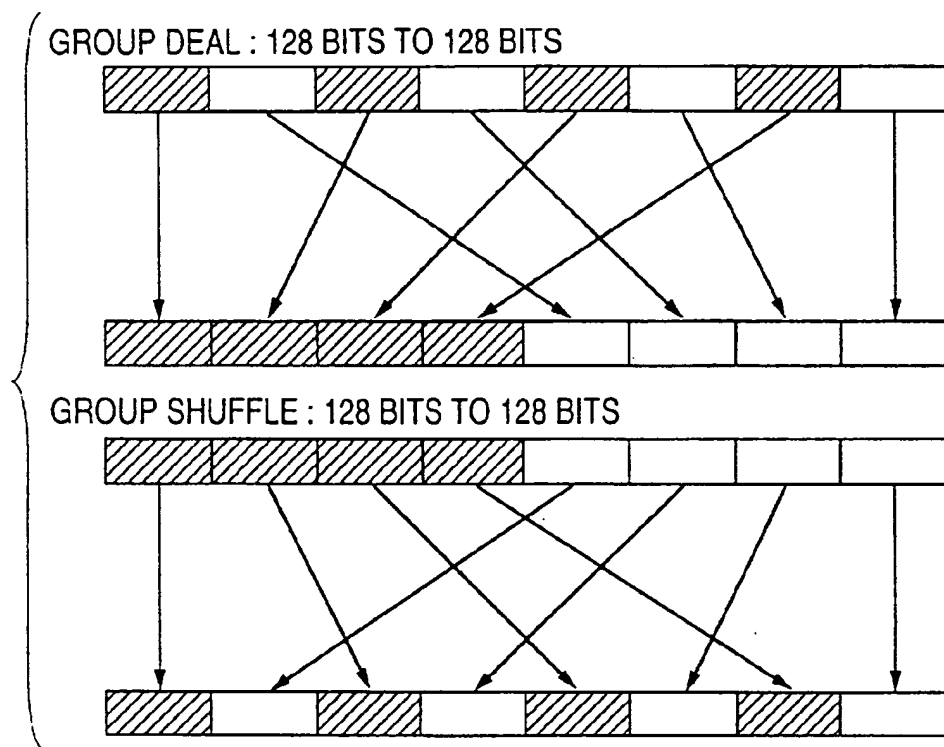


FIG. 8(d)

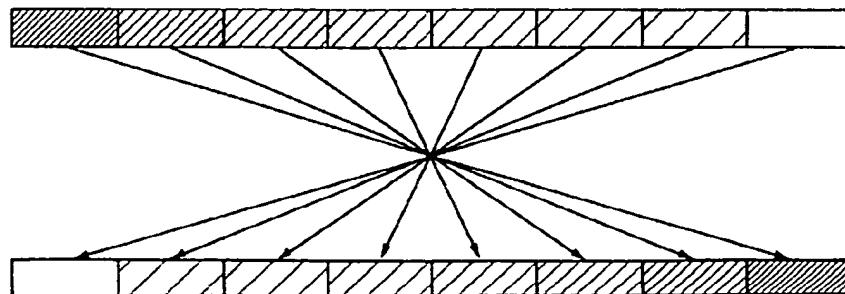
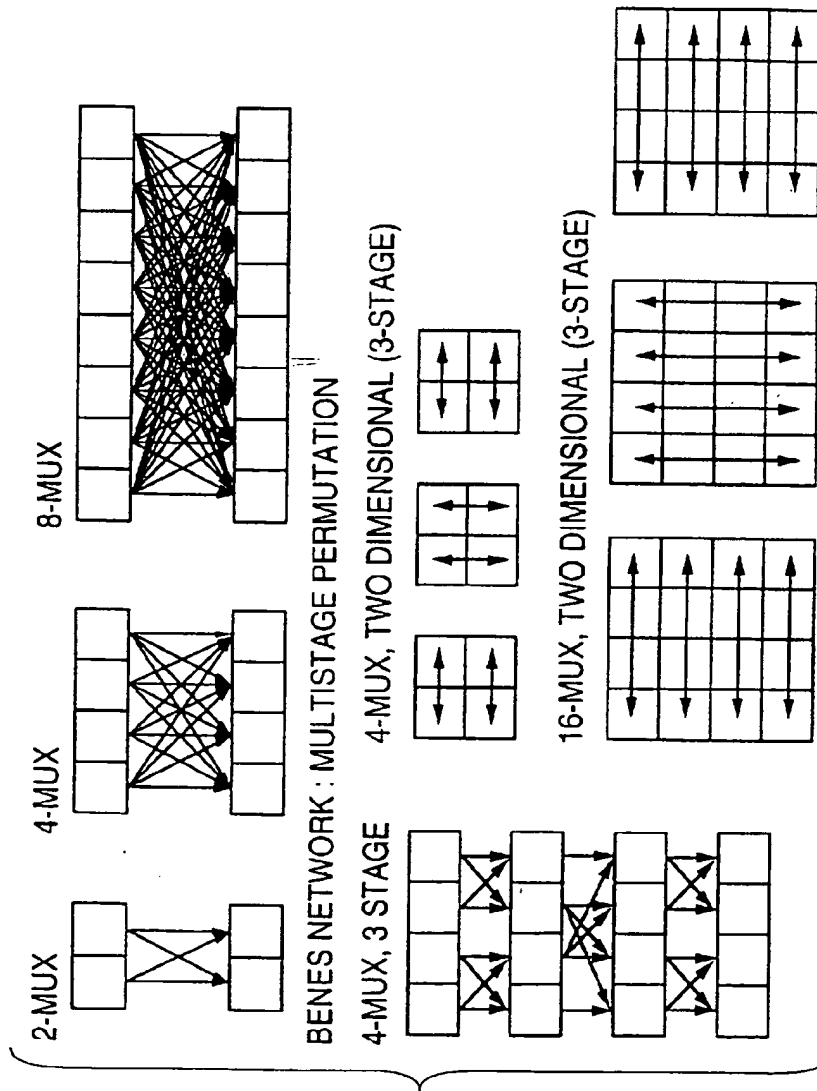


FIG. 8(e)



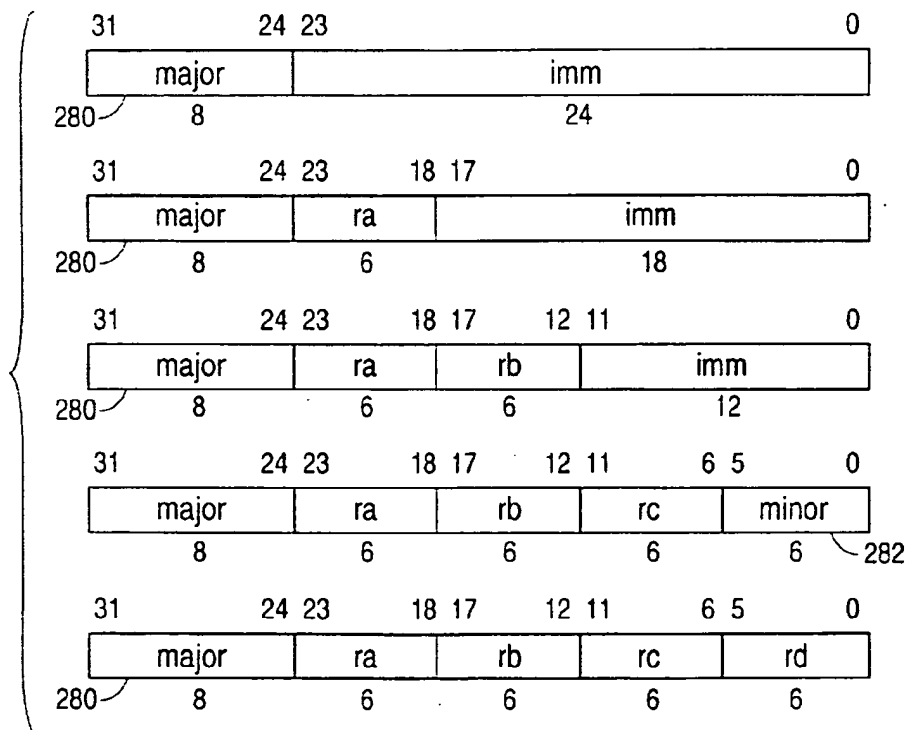
U.S. Patent

Sep. 15, 1998

Sheet 11 of 25

5,809,321

FIG. 9(a)



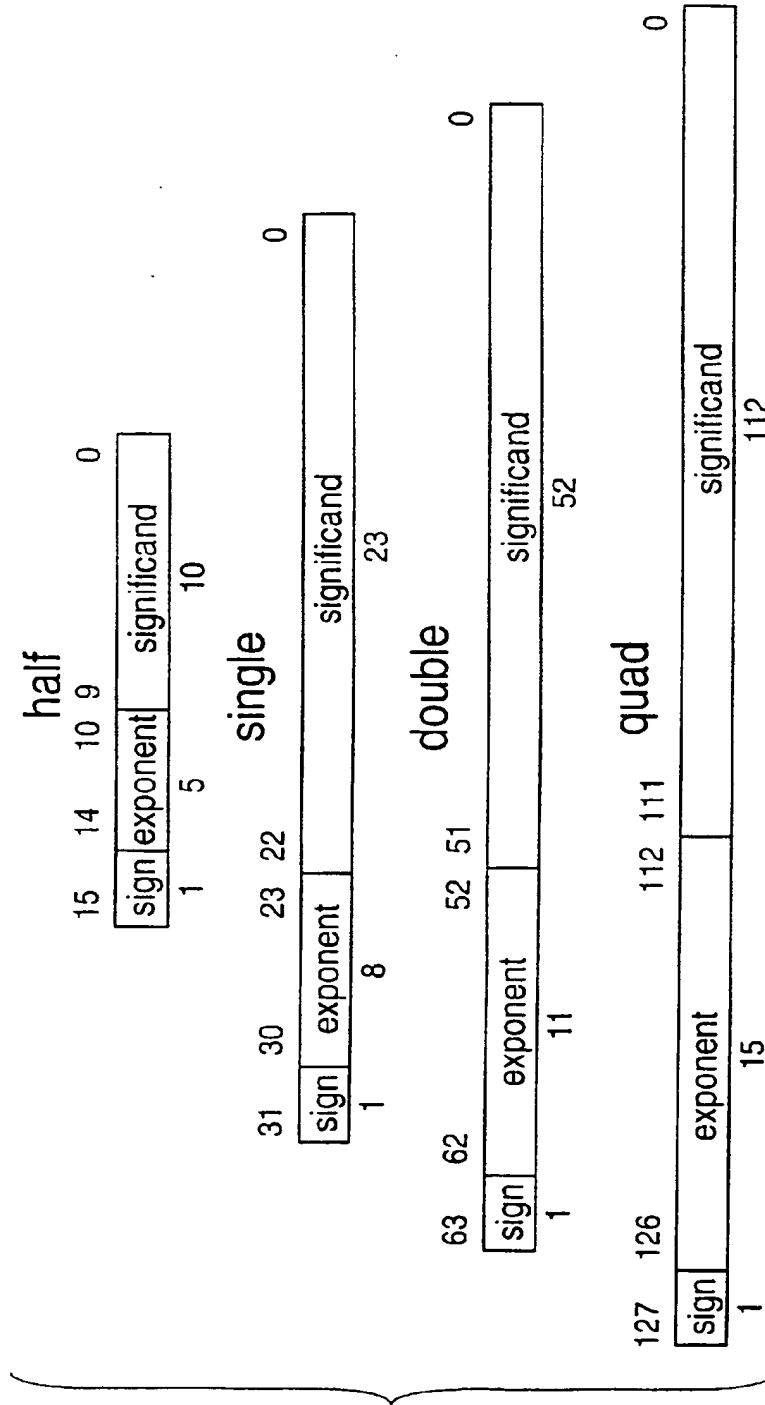
U.S. Patent

Sep. 15, 1998

Sheet 12 of 25

5,809,321

FIG. 9(b)



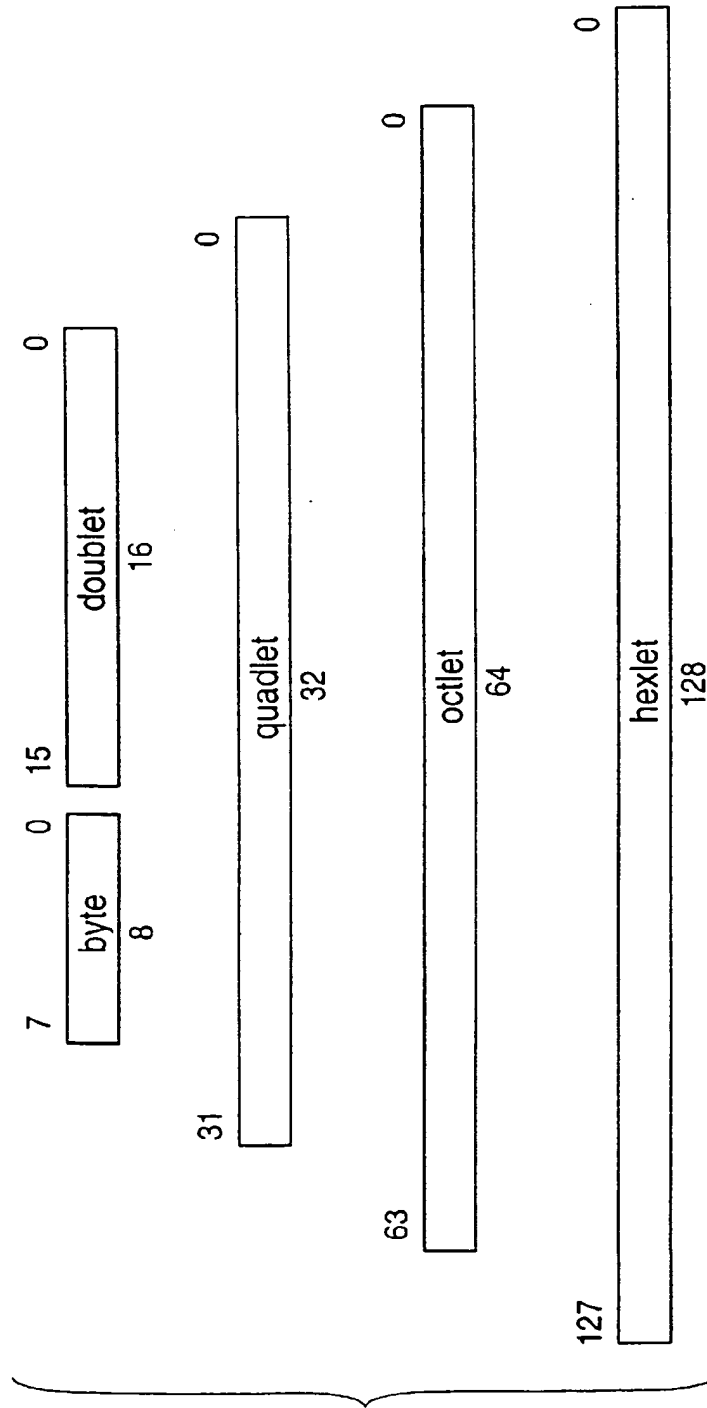
U.S. Patent

Sep. 15, 1998

Sheet 13 of 25

5,809,321

FIG. 9(c)



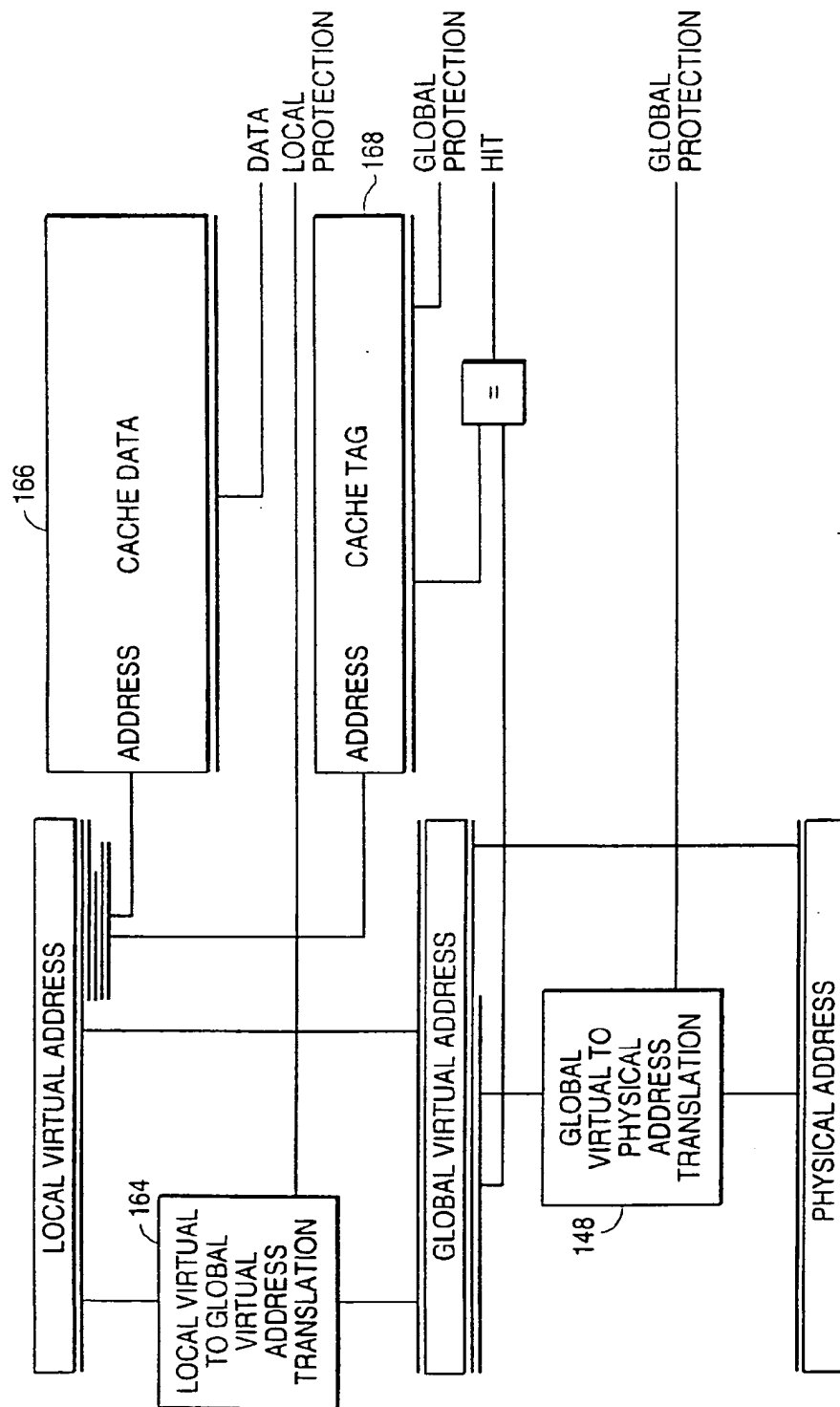
U.S. Patent

Sep. 15, 1998

Sheet 14 of 25

5,809,321

FIG. 10(a)

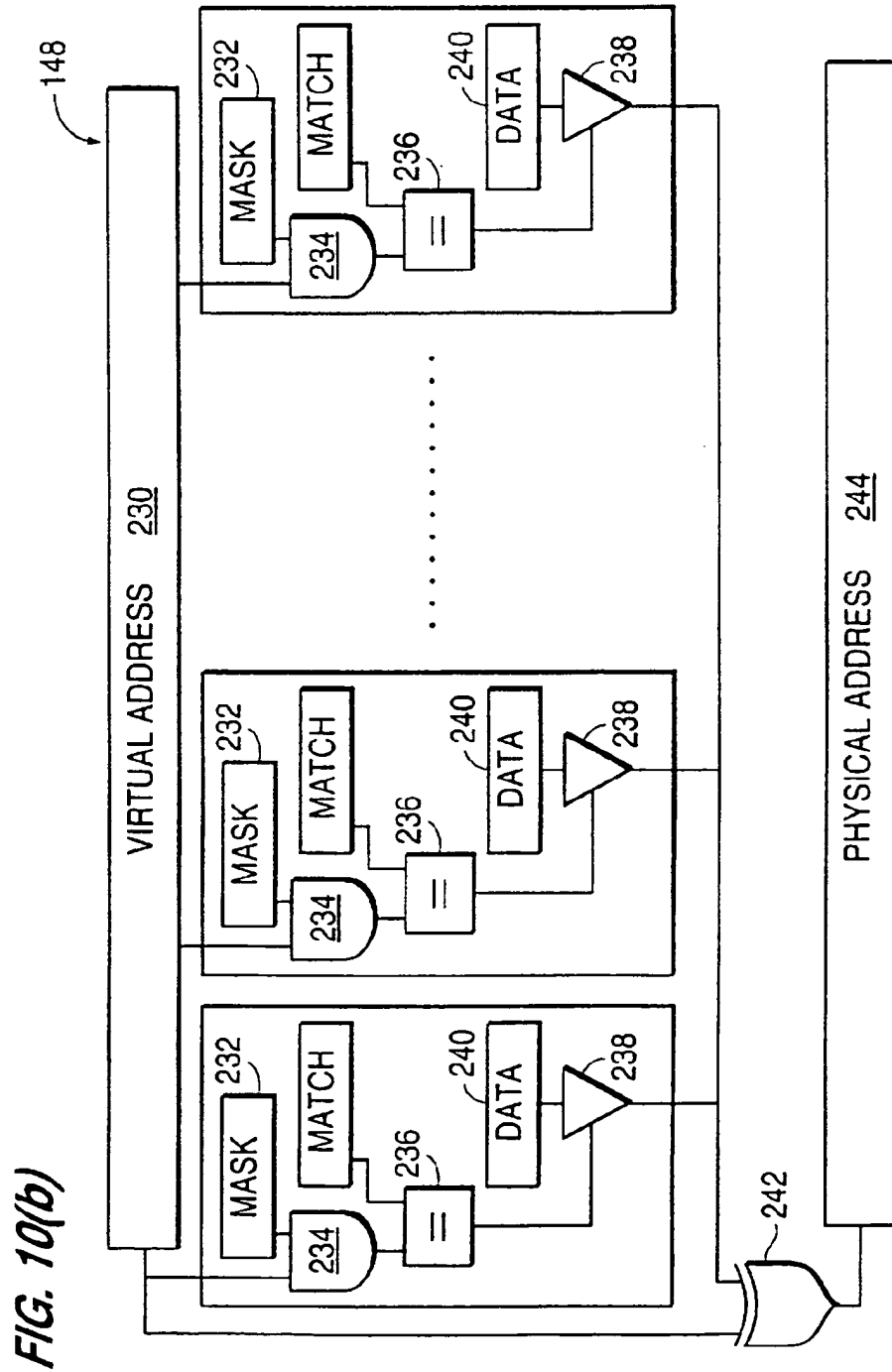


U.S. Patent

Sep. 15, 1998

Sheet 15 of 25

5,809,321



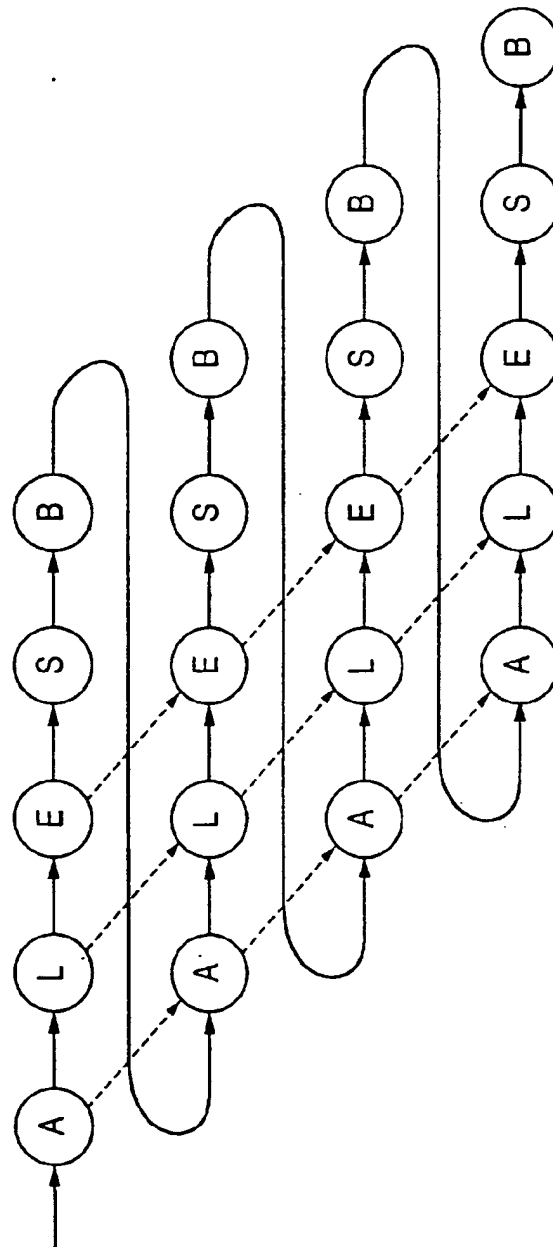
U.S. Patent

Sep. 15, 1998

Sheet 16 of 25

5,809,321

FIG. 11

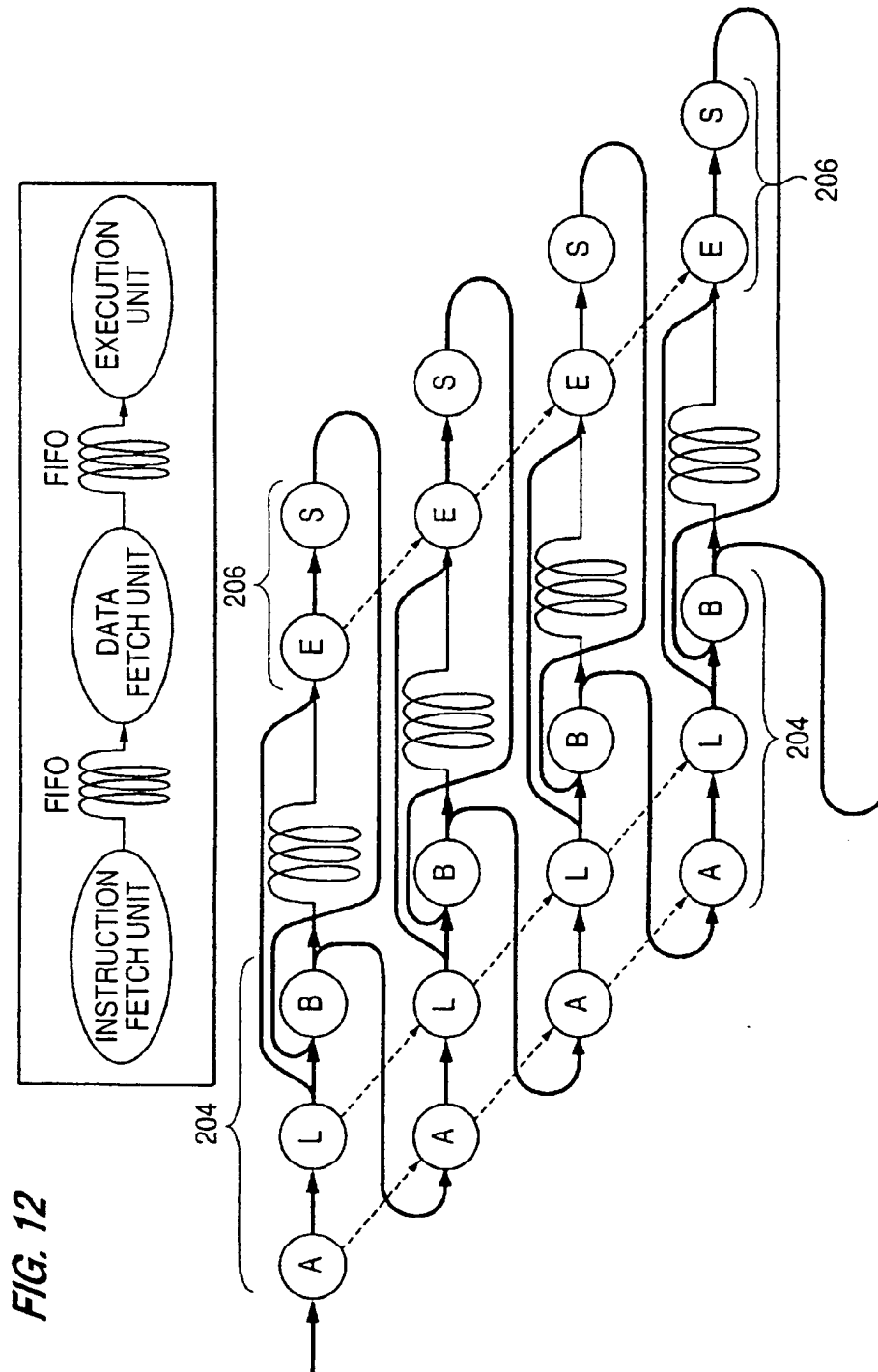


U.S. Patent

Sep. 15, 1998

Sheet 17 of 25

5,809,321



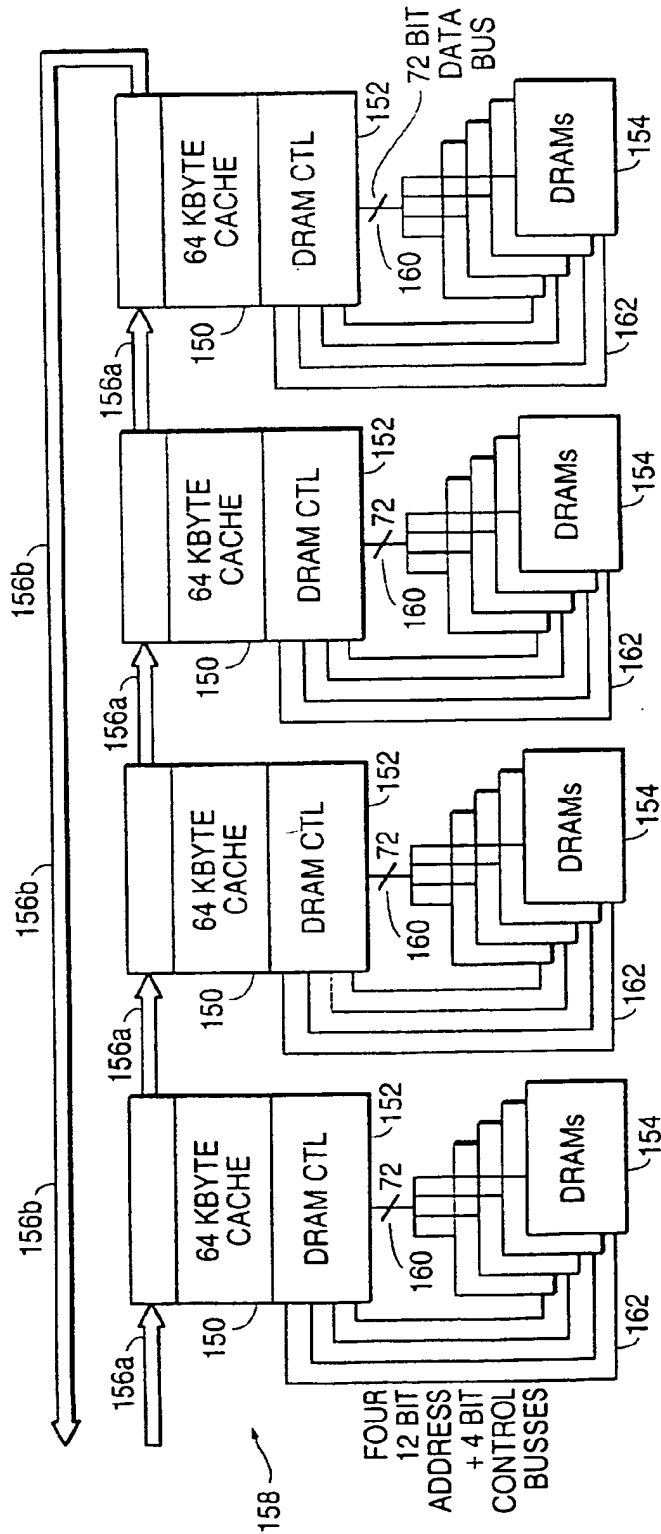
U.S. Patent

Sep. 15, 1998

Sheet 18 of 25

5,809,321

FIG. 13

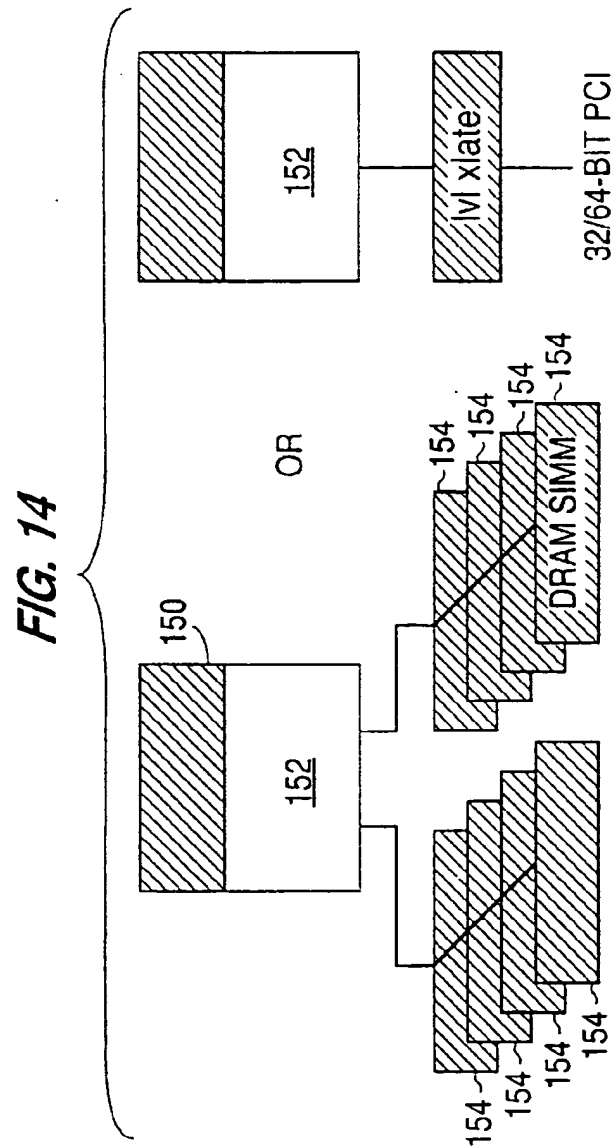


U.S. Patent

Sep. 15, 1998

Sheet 19 of 25

5,809,321



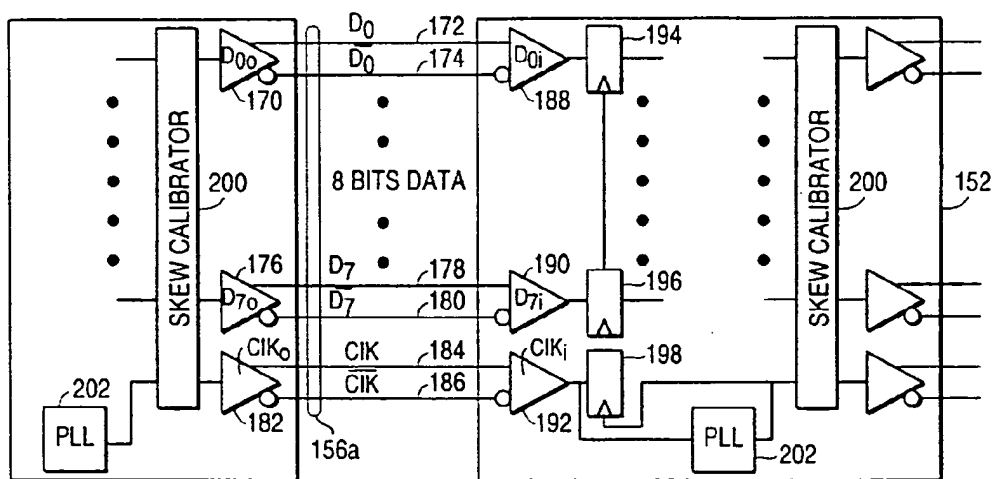
U.S. Patent

Sep. 15, 1998

Sheet 20 of 25

5,809,321

FIG. 15



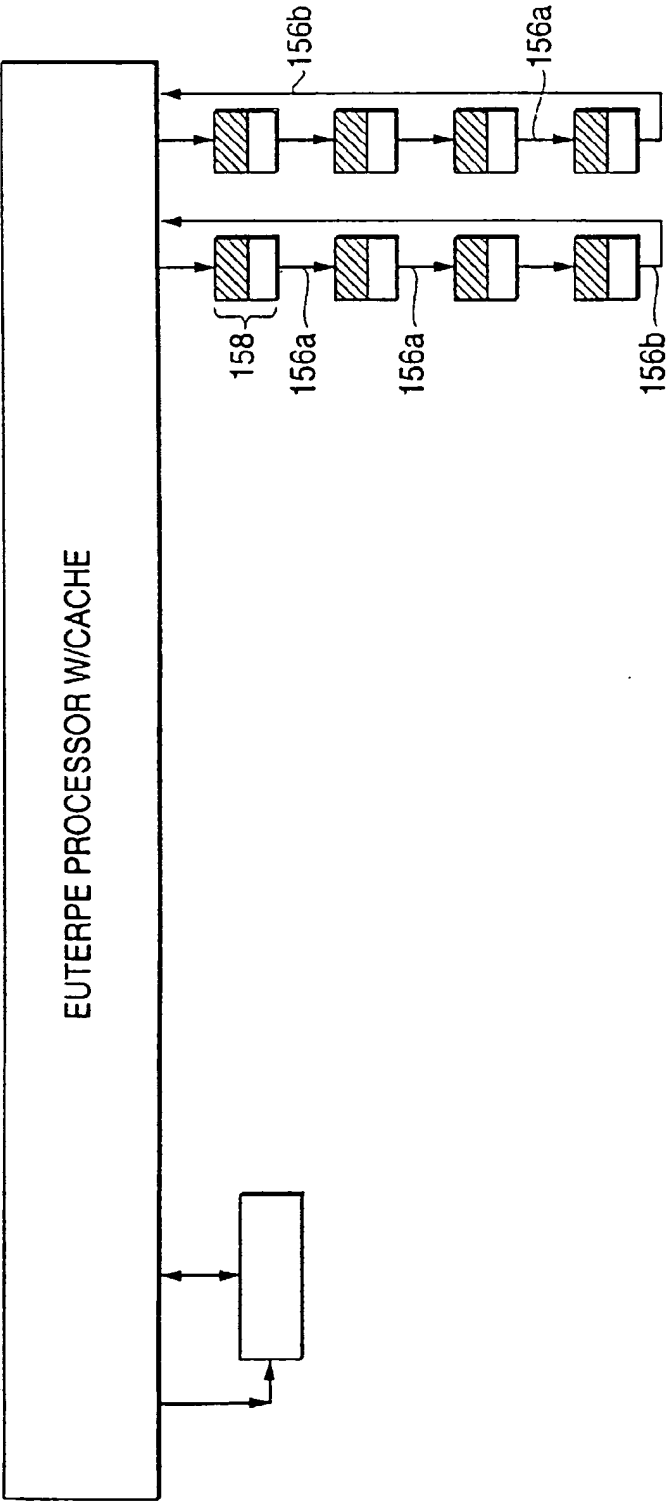
U.S. Patent

Sep. 15, 1998

Sheet 21 of 25

5,809,321

FIG. 16(a)



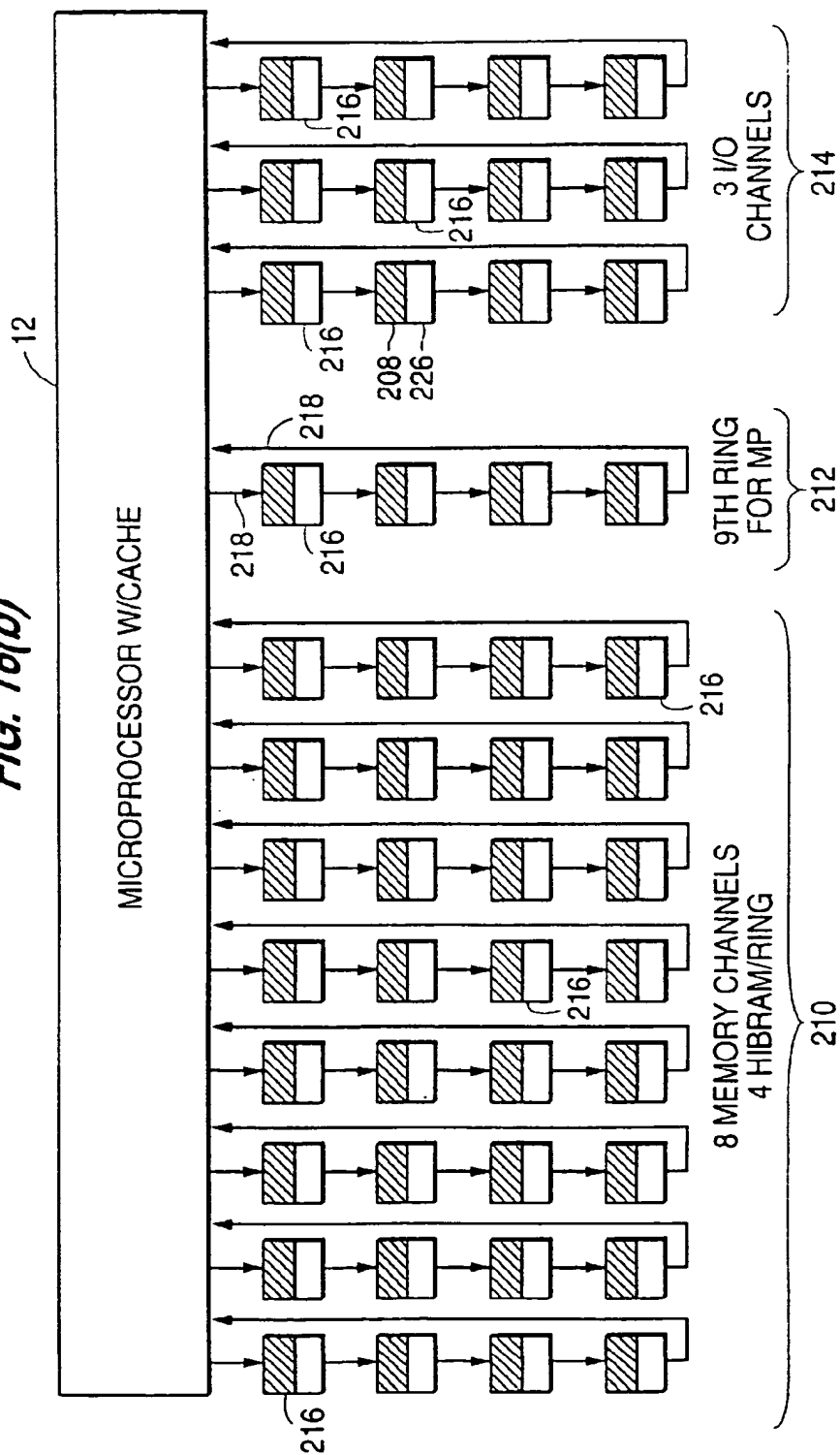
U.S. Patent

Sep. 15, 1998

Sheet 22 of 25

5,809,321

FIG. 16(b)

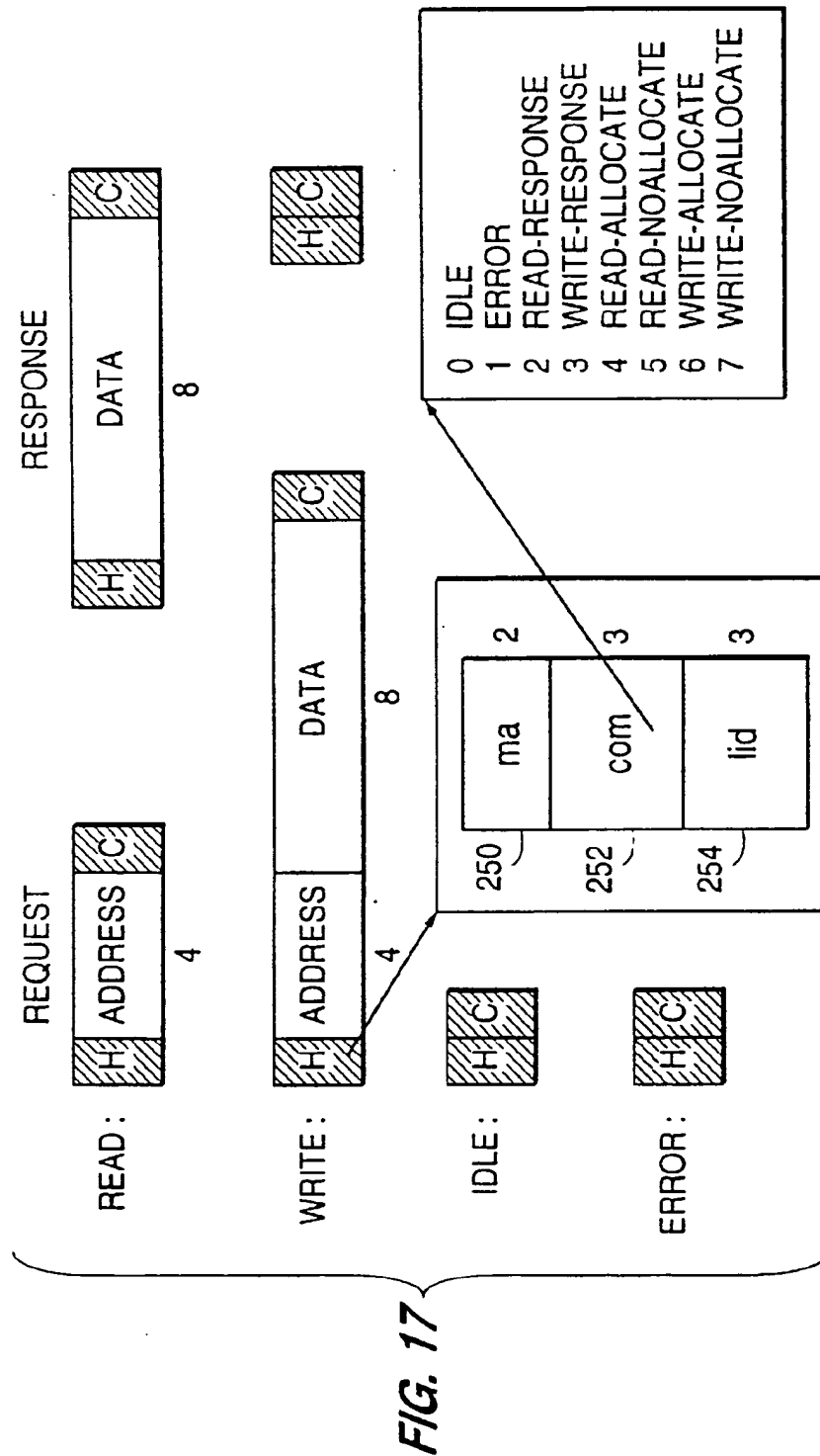


U.S. Patent

Sep. 15, 1998

Sheet 23 of 25

5,809,321



U.S. Patent

Sep. 15, 1998

Sheet 24 of 25

5,809,321

FIG. 18(a)

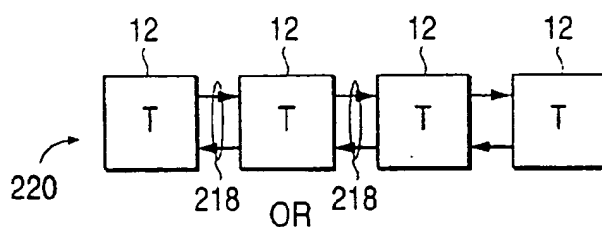


FIG. 18(b)

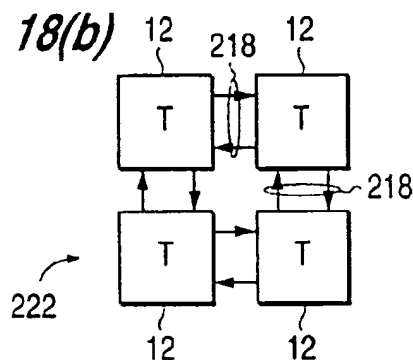
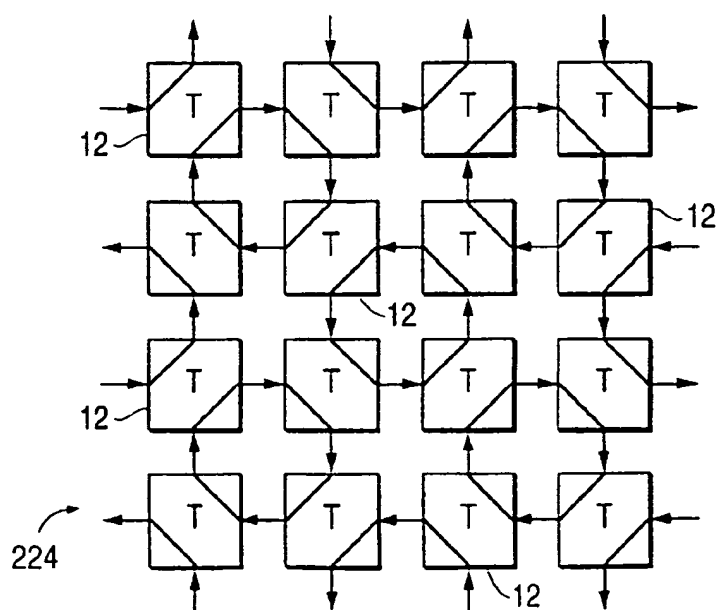


FIG. 18(c)



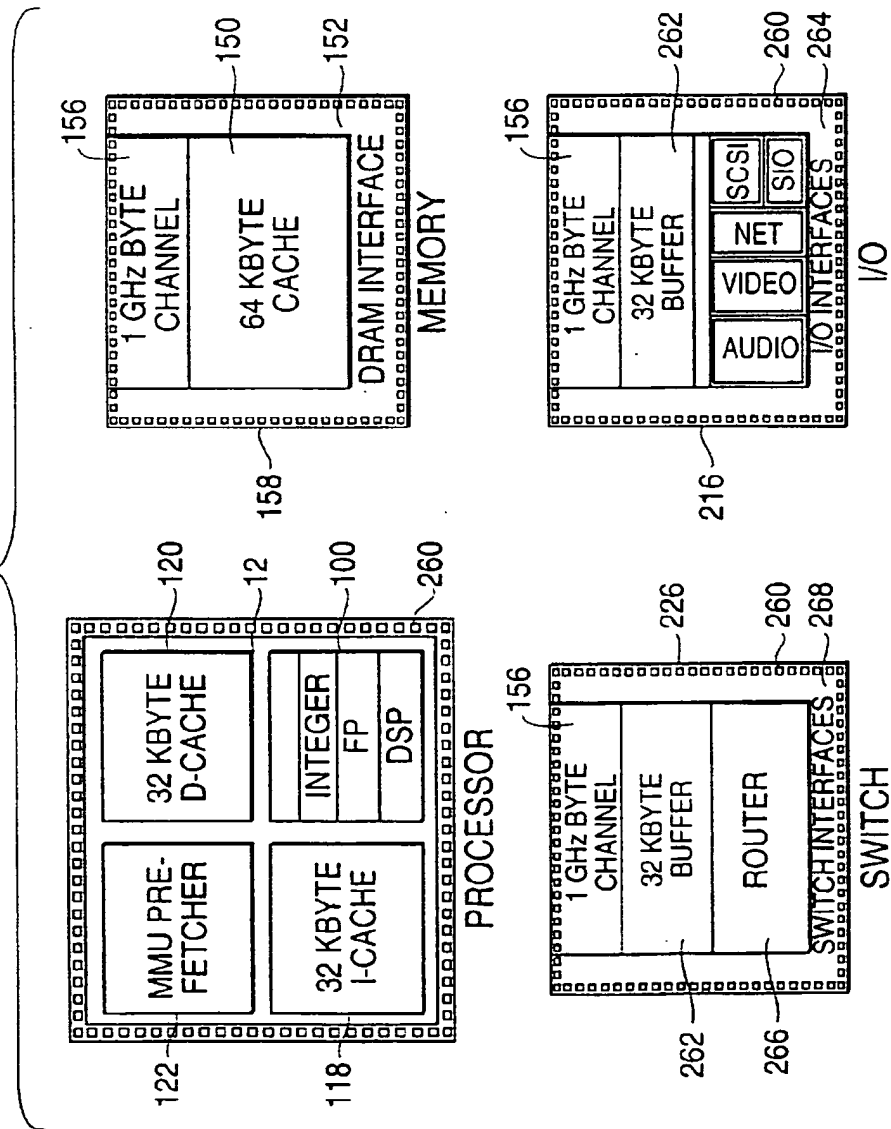
U.S. Patent

Sep. 15, 1998

Sheet 25 of 25

5,809,321

FIG. 19



5,809,321

1

GENERAL PURPOSE, MULTIPLE PRECISION PARALLEL OPERATION, PROGRAMMABLE MEDIA PROCESSOR

This is a divisional of application Ser. No. 08/516,036, filed Aug. 16, 1995, now U.S. Pat. No. 5,742,840.

A Microfiche Appendix consisting of 4 sheets (387 total frames) of microfiche is included in this application. The Microfiche Appendix contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by any one of the Microfiche Appendix, as it appears in the Patent and Trademark Office patent files or records, but otherwise reserves all copyright rights whatsoever.

FIELD OF THE INVENTION

This invention relates to the field of communications processing, and more particularly, to a method and apparatus for real-time processing of multi-media digital communications.

BACKGROUND OF THE INVENTION

Optical fiber and discs have made the transmission and storage of digital information both cheaper and easier than older analog technologies. An improved system for digital processing of media data streams is necessary in order to realize the full potential of these advanced media.

For the past century, telephone service delivered over copper twisted pair has been the lingua franca of communications. Over the next century, broadband services delivered over optical fiber and coax will more completely fulfill the human need for sensory information by supplying voice, video, and data at rates of about 1,000 times greater than narrow band telephony. Current general-purpose microprocessors and digital signal processors ("DSPs") can handle digital voice, data, and images at narrow band rates, but they are way too slow for processing media data at broadband rates.

This shortfall in digital processing of broadband media is currently being addressed through the design of many different kinds of application-specific integrated circuits ("ASICs"). For example, a prototypical broadband device such as a cable modem modulates and demodulates digital data at rates up to 45 Mbits/sec within a single 6 MHz cable channel (as compared to rates of 28.8 Kbits/sec within a 6 KHz channel for telephone modems) and transcodes it onto a 10/100 baseT connection to a personal computer ("PC") or workstation. Current cable modems thus receive data from a coaxial cable connection through a chain of specialized ASIC devices in order to accomplish Quadrature Amplitude Modulation ("QAM") demodulation, Reed-Solomon error correction, packet filtering, Data Encryption Standard ("DES") decryption, and Ethernet protocol handling. The cable modems also transmit data to the coaxial cable link through a second chain of devices to achieve DES encryption, Reed-Solomon block encoding, and Quaternary Phase Shift Keying ("QPSK") modulation. In these environments, a general-purpose processor is usually required as well in order to perform initialization, statistics collection, diagnostics, and network management functions.

The ASIC approach to media processing has three fundamental flaws: cost, complexity, and rigidity. The combined silicon area of all the specialized ASIC devices required in the cable modem, for example, results in a component cost incompatible with the per subscriber price target for a cable service. The cable plant itself is a very

2

hostile service environment, with noise ingress, reflections, nonlinear amplifiers, and other channel impairments, especially when viewed in the upstream direction. Telephony modems have developed an elaborate hierarchy of algorithms implemented in DSP software, with automatic reduction of data rates from 28.8 Kbits/sec to 19.6 Kbits/sec, 14.4 Kbits/sec, or much lower rates as needed to accommodate noise, echoes, and other impairments in the copper plant. To implement similar algorithms on an ASIC-based broadband modem is far more complex to achieve in software.

These problems of cost, complexity, and rigidity are compounded further in more complete broadband devices such as digital set-top boxes, multimedia PCs, or video conferencing equipment, all of which go beyond the basic radio frequency ("RF") modem functions to include a broad range of audio and video compression and decoding algorithms, along with remote control and graphical user interfaces. Software for these devices must control what amounts to a heterogeneous multi-processor, where each specialized processor has a different, and usually eccentric or primitive, programming environment. Even if these programming environments are mastered, the degree of programmability is limited. For example, Motion Picture Expert Group-I ("MPEG-I") chips manufactured by AT&T Corporation will not implement advances such as fractal- and wavelet-based compression algorithms, but these chips are not readily software upgradeable to the MPEG-II standard. A broadband network operator who leases an MPEG ASIC-based product is therefore at risk of having to continuously upgrade his system by purchasing significant amounts of new hardware just to track the evolution of MPEG standards.

The high cost of ASIC-based media processing results from inefficiencies in both memory and logic. A typical ASIC consists of a multiplicity of specialized logic blocks, each with a small memory dedicated to holding the data which comprises the working set for that block. The silicon area of these multiple small memories is further increased by the overhead of multiple decoders, sense amplifiers, write drivers, etc. required for each logic block. The logic blocks are also constrained to operate at frequencies determined by the internal symbol rates of broadband algorithms in order to avoid additional buffer memories. These frequencies typically differ from the optimum speed-area operating point of a given semiconductor technology. Interconnect and synchronization of the many logic and memory blocks are also major sources of overhead in the ASIC approach.

The disadvantages of the prior ASIC approach can be overcome by a single unified media processor. The cost advantages of such a unified processor can be achieved by gathering all the many ASIC functions of a broadband media product into a single integrated circuit. Cost reduction is further increased by reducing the total memory area of such a circuit by replacing the multiplicity of small ASIC memories with a single memory hierarchy large enough to accommodate the sum total of all the working sets, and wide enough to supply the aggregate bandwidth needs of all the logic blocks. Additionally, the logic block interconnect circuitry to this memory hierarchy may be streamlined by providing a generally programmable switching fabric. Many of the logic blocks themselves can also be replaced with a single multi-precision arithmetic unit, which can be internally partitioned under software control to perform addition, multiplication, division, and other integer and floating point arithmetic operations on symbol streams of varying widths, while sustaining the full data throughput of the memory hierarchy. The residue of logic blocks that perform opera-

5,809,321

3

tions that are neither arithmetic or permutation group oriented can be replaced with an extended math unit that supports additional arithmetic operations such as finite field, ring, and table lookup, while also sustaining the full data throughput of the memory hierarchy.

The above multi-precision arithmetic, permutation switch, and extended math operations can then be organized as machine instructions that transfer their operands to and from a single wide multi-ported register file. These instructions can be further supplemented with load/store instructions that transfer register data to and from a data buffer/cache static random access memory ("SRAM") and main memory dynamic random access memories ("DRAMs"), and with branch instructions that control the flow of instructions executed from an instruction buffer/cache SRAM. Extensions to the load/store instructions can be made for synchronization, and to branch instructions for protected gateways, so that multiple threads of execution for audio, video, radio, encryption, networking, etc. can efficiently and securely share memory and logic resources of a unified machine operating near the optimum speed-area point of the target semiconductor process. The data path for such a unified media processor can interface to a high speed input/output ("I/O") subsystem that moves media streams across ultra-high bandwidth interfaces to external storage and I/O.

Such a device would incorporate all of the processing capabilities of the specialized multi-ASIC combination into a single, unified processing device. The unified processor would be agile and capable of reprogramming through the transmission of new programs over the communication medium. This programmable, general purpose device is thus less costly than the specialized processor combination, easier to operate and reprogram and can be installed or applied in many differing devices and situations. The device may also be scalable to communications applications that support vast numbers of users through massively parallel distributed computing.

It is therefore an object of this invention to process media data streams by executing operations at very high bandwidth rates.

It is also an object of this invention to unify the audio, video, radio, graphics, encryption, authentication, and networking protocols into a single instruction stream.

It is also an object of this invention to achieve high bandwidth rates in a unified processor that is easy to program and more flexible than a heterogeneous combination of special purpose processors.

It is a further object of the invention to support high level mathematical processing in a unified media processor, including finite group, finite field, finite ring and table look-up operations, all at high bandwidth rates.

It is yet a further object of the invention to provide a unified media processor that can be replicated into a multi-processor system to support a vast array of users.

It is yet another object of this invention to allow for massively parallel systems within the switching fabric to support very large numbers of subscribers and services.

It is also an object of the invention to provide a general purpose programmable processor that could be employed at all points in a network.

It is a further object of this invention to sustain very high bandwidth rates to arbitrarily large memory and input/output systems.

SUMMARY OF THE INVENTION

In view of the above, there is provided a system for media processing that maintains substantially peak data throughput

4

in the execution and transmission of multiple media data streams. The system includes in one aspect a general purpose, programmable media processor, and in another aspect includes a method for receiving, processing and transmitting media data streams. The general purpose, programmable media processor of the invention further includes an execution unit, high bandwidth external interface, and can be employed in a parallel multi-processor system.

According to the apparatus of the invention, an execution unit is provided that maintains substantially peak data throughput in the unified execution of multiple media data streams. The execution unit includes a data path, and a multi-precision arithmetic unit coupled to the data path and capable of dynamic partitioning based on the elemental width of data received from the data path. The execution unit also includes a switch coupled to the data path that is programmable to manipulate data received from the data path and provide data streams to the data path. An extended mathematical element is also provided, which is coupled to the data path and programmable to implement additional mathematical operations at substantially peak data throughput. In a preferred embodiment of the execution unit, at least one register file is coupled to the data path.

According to another aspect of the invention, a general purpose programmable media processor is provided having an instruction path and a data path to digitally process a plurality of media data streams. The media processor includes a high bandwidth external interface operable to receive a plurality of data of various sizes from an external source and communicate the received data over the data path at a rate that maintains substantially peak operation of the media processor. At least one register file is included, which is configurable to receive and store data from the data path and to communicate the stored data to the data path. A multi-precision execution unit is coupled to the data path and is dynamically configurable to partition data received from the data path to account for the elemental symbol size of the plurality of media streams, and is programmable to operate on the data to generate a unified symbol output to the data path.

According to the preferred embodiment of the media processor, means are included for moving data between registers and memory by performing load and store operations, and for coordinating the sharing of data among a plurality of tasks by performing synchronization operations based upon instructions and data received by the execution unit. Means are also provided for securely controlling the sequence of execution by performing branch and gateway operations based upon instructions and data received by the execution unit. A memory management unit operable to retrieve data and instructions for timely and secure communication over the data path and instruction path respectively is also preferably included in the media processor. The preferred embodiment also includes a combined instruction cache and buffer that is dynamically allocated between cache space and buffer space to ensure real-time execution of multiple media instruction streams, and a combined data cache and buffer that is dynamically allocated between cache space and buffer space to ensure real-time response for multiple media data streams.

In another aspect of the invention, a high bandwidth processor interface for receiving and transmitting a media stream is provided having a data path operable to transmit media information at sustained peak rates. The high bandwidth processor interface includes a plurality of memory controllers coupled in series to communicate stored media

5,809,321

5

information to and from the data path, and a plurality of memory elements coupled in parallel to each of the plurality of memory controllers for storing and retrieving the media information. In the preferred embodiment of the high bandwidth processor interface, the plurality of memory controllers each comprise a paired link disposed between each memory controller, where the paired links each transmit and receive plural bits of data and have differential data inputs and outputs and a differential clock signal.

Yet another aspect of the invention includes a system for unified media processing having a plurality of general purpose media processors, where each media processor is operable at substantially peak data rates and has a dynamically partitioned execution unit and a high bandwidth interface for communicating to memory and input/output elements to supply data to the media processor at substantially peak rates. A bi-directional communication fabric is provided, to which the plurality of media processors are coupled, to transmit and receive at least one media stream comprising presentation, transmission, and storage media information. The bi-directional communication fabric preferably comprises a fiber optic network, and a subset of the plurality of media processors comprise network servers.

According to yet another aspect of the invention, a parallel multi-media processor system is provided having a data path and a high bandwidth external interface coupled to the data path and operable to receive a plurality of data of various sizes from an external source and communicate the received data at a rate that maintains substantially peak operation of the parallel multi-processor system. A plurality of register files, each having at least one register coupled to the data path and operable to store data, are also included. At least one multiprecision execution unit is coupled to the data path and is dynamically configurable to partition data received from the data path to account for the elemental symbol size of the plurality of media streams, and is programmable to operate in parallel on data stored in the plurality of register files to generate a unified symbol output for each register file.

According to the method of the invention, unified streams of media data are processed by receiving a stream of unified media data including presentation, transmission and storage information. The unified stream of media data is dynamically partitioned into component fields of at least one bit based on the elemental symbol size of data received. The unified stream of media data is then processed at substantially peak operation.

In one aspect of the invention, the unified stream of media data is processed by storing the stream of unified media data in a general register file. Multi-precision arithmetic operations can then be performed on the stored stream of unified media data based on programmed instructions, where the multi-precision arithmetic operations include Boolean, integer and floating point mathematical operations. The component fields of unified media data can then be manipulated based on programmed instructions that implement copying, shifting and re-sizing operations. Multi-precision mathematical operations can also be performed on the stored stream of unified media data based on programmed instructions, where the mathematical operations including finite group, finite field, finite ring and table look-up operations. Instruction and data pre-fetching are included to fill instruction and data pipelines, and memory management operations can be performed to retrieve instructions and data from external memory. The instructions and data are preferably stored in instruction and data cache/buffers, in which buffer storage in the instruction and data cache/buffers is dynamically allocated to ensure real-time execution.

6

Other aspects of the invention include a method for achieving high bandwidth communications between a general purpose media processor and external devices by providing a high bandwidth interface disposed between the media processor and the external devices, in which the high bandwidth interface comprises at least one uni-directional channel pair having an input port and an output port. A plurality of media data streams, comprising component fields of various sizes, are transmitted and received between the media processor and the external devices at a rate that sustains substantially peak data throughput at the media processor. A method for processing streams of media data is also included that provides a bi-directional communications fabric for transmitting and receiving at least one stream of media data, where the at least one stream of media data comprises presentation, transmission and storage information. At least one programmable media processor is provided within the communications network for receiving, processing and transmitting the at least one stream of unified media data over the bi-directional communications fabric.

The general purpose, programmable media processor of the invention combines in a single device all of the necessary hardware included in the specialized processor combinations to process and communicate digital media data streams in real-time. The general purpose, programmable media processor is therefore cheaper and more flexible than the prior approach to media processing. The general purpose, programmable media processor is thus more susceptible to incorporation within a massively parallel processing network of general purpose media processors that enhance the ability to provide real-time multi-media communications to the masses.

These features are accomplished by deploying server media processors and client media processors throughout the network. Such a network provides a seamless, global media super-computer which allows programmers and network owners to virtualize resources. Rather than restrictively accessing only the memory space and processing time of a local resource, the system allows access to resources throughout the network. In small access points such as wireless devices, where very little memory and processing logic is available due to limited battery life, the system is able to draw upon the resources of a homogeneous multi-computer system.

The invention also allows network owners the facility to track standards and to deploy new services by broadcasting software across the network rather than by instituting costly hardware upgrades across the whole network. Broadcasting software across the network can be performed at the end of an advertisement or other program that is broadcasted nationally. Thus, services can be advertised and then transmitted to new subscribers at the end of the advertisement.

These and other features and advantages of the invention will be apparent upon consideration of the following detailed description of the presently preferred embodiments of the invention, taken in conjunction with the appended drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a broad band media computer employing the general purpose, programmable media processor of the invention;

FIG. 2 is a block diagram of a global media processor employing multiple general purpose media processors according to the invention;

FIG. 3 is an illustration of the digital bandwidth spectrum for telecommunications, media and computing communications;

5,809,321

7

FIG. 4 is the digital bandwidth spectrum shown in FIG. 3 taking into account the bandwidth overhead associated with compressed video techniques;

FIG. 5 is a block diagram of the current specialized processor solution for mass media communication, where FIG. 5(a) shows the current distributed system, and FIG. 5(b) shows a possible integrated approach;

FIG. 6 is a block diagram of two presently preferred general purpose media processors, where FIG. 6(a) shows a distributed system and FIG. 6(b) shows an integrated media processor;

FIG. 7 is a block diagram of the presently preferred structure of a general purpose, programmable media processor according to the invention;

FIG. 8 is a drawing consisting of visual illustrations of the various group operations provided on the media processor, where FIG. 8(a) illustrates the group expand operation, FIG. 8(b) illustrates the group compress or extract operation, FIG. 8(c) illustrates the group deal and shuffle operations, FIG. 8(d) illustrates the group swizzle operation and FIG. 8(e) illustrates the various group permute operations;

FIG. 9 shows the preferred instruction and data sizes for the general purpose, programmable media processor, where FIG. 9(a) is an illustration of the various instruction formats available on the general purpose, programmable media processor, FIG. 9(b) illustrates the various floating-point data sizes available on the general purpose media processor, and FIG. 9(c) illustrates the various fixedpoint data sizes available on the general purpose media processor;

FIG. 10 is an illustration of a presently preferred memory management unit included in the general purpose processor shown in FIG. 7, where FIG. 10(a) is a translation block diagram and FIG. 10(b) illustrates the functional blocks of the transaction lookaside buffer;

FIG. 11 is an illustration of a super-string pipeline technique;

FIG. 12 is an illustration of the presently preferred super-string pipeline technique;

FIG. 13 is a block diagram of a single memory channel for communication to the general purpose media processor shown in FIG. 7;

FIG. 14 is an illustration of the presently preferred connection of standard memory devices to the preferred memory interface;

FIG. 15 is a block diagram of the input/output controller for use with the memory channel shown in FIG. 13;

FIG. 16 is a block diagram showing multiple memory channels connected to the general purpose media processor shown in FIG. 7, where FIG. 16(a) shows a two-channel implementation and FIG. 16(b) illustrates a twelve-channel channel embodiment;

FIG. 17 illustrates the presently preferred packet communications protocol for use over the memory channel shown in FIG. 13;

FIG. 18 shows a multi-processor configuration employing the general purpose media processor shown in FIG. 7, where FIG. 18(a) shows a linear processor configuration, FIG. 18(b) shows a processor ring configuration, and FIG. 18(c) shows a two-dimensional processor configuration; and

FIG. 19 shows a presently preferred multi-chip implementation of the general purpose, programmable media processor of the invention.

DETAILED DESCRIPTION OF THE PRESENTLY PREFERRED EMBODIMENT

Referring to the drawings, where like-reference numerals refer to like elements throughout, a broad band microcom-

8

puter 10 is provided in FIG. 1. The broad band microcomputer 10 consists essentially of a general purpose media processor 12. As will be described in more detail below, the general purpose media processor 12 receives, processes and transmits media data streams in a bi-directional manner from upstream network components to downstream devices. In general, media data streams received from upstream network components can comprise any combination of audio, video, radio, graphics, encryption, authentication, and networking information. As those skilled in the art will appreciate, however, the general purpose media processor 12 is in no way limited to receiving, processing and transmitting only these types of media information. The general purpose media processor 12 of the invention is capable of processing any form of digital media information without departing from the spirit and essential scope of the invention.

System Configuration

In the preferred embodiment of the invention shown in FIG. 1, media data streams are communicated to the media processor 12 from several sources. Ideally, unified media data streams are received and transmitted by the general purpose media processor 12 over a fiber optic cable network 14. As will be described in more detail below, although a fiber optic cable network is preferred, the presently existing communications network in the United States consists of a combination of fiber optic cable, coaxial cable and other transmission media. Consequently, the general purpose media processor 12 can also receive and transmit media data streams over coaxial cable 14 and traditional twisted pair wire connections 16. The specific communications protocol employed over the twisted pair 16, whether POTS, ISDN or ADSL, is not essential; all protocols are supported by the broad band microcomputer 10. The details of these protocols are generally known to those skilled in the art and no further discussion is therefore needed or provided herein.

Another form of upstream network communication is through a satellite link 18. The satellite link 18 is typically connected to a satellite receiver 20. The satellite receiver 20 comprises an antenna, usually in the form of a satellite dish, and amplification circuitry. The details of such satellite communications are also generally known in the art, and further detail is therefore not provided or included herein.

As described above, the general purpose media processor 12 communicates in a bi-directional manner to receive, process and transmit media data streams to and from downstream devices. As shown in FIG. 1, downstream communication preferably takes place in at least two forms. First, media data streams can be communicated over a bi-directional local network 22. Various types of local networks 22 are generally known in the art and many different forms exist. The general purpose media processor 12 is capable of communicating over any of these local networks 22 and the particular type of network selected is implementation specific.

The local network 22 is preferably employed to communicate between the unified processor 12, and audio/visual devices 24 or other digital devices 26. Presently preferred examples of audio/visual devices 24 include digital cable television, video-on-demand devices, electronic yellow pages services, integrated message systems, video telephones, video games and electronic program guides. As those skilled in the art will appreciate, other forms of audio/video devices are contemplated within the spirit and scope of the invention. Presently preferred embodiments of other digital devices 26 for communication with the general purpose media processor 12 include personal computers,

5,809,321

9

television sets, work stations, digital video camera recorders, and compact disc read-only memories. As those skilled in the art will also appreciate, further digital devices 26 are contemplated for communication to the general purpose media processor 12 without departing from the spirit and scope of the invention.

Second, the general purpose media processor preferably also communicates with downstream devices over a wireless network 28. In the presently preferred embodiment of the invention, wireless devices for communication over the wireless network 28 can comprise either remote communication devices 30 or remote computing devices 32. Presently preferred embodiments of the remote communications devices 30 include cordless telephones and personal communicators. Presently preferred embodiments of the remote computing devices 32 include remote controls and telecommunicating devices. As those skilled in the art will appreciate, other forms of remote communication devices 30 and remote computing devices 32 are capable of communication with the general purpose media processor 12 without departing from the spirit and scope of the invention. An agile digital radio (not shown) that incorporates a general purpose media processor 12 may be used to communicate with these wireless devices.

Network Configuration

Referring now to FIG. 2, the general purpose media processor 12 is preferably disposed throughout a digital communications network 38. In order to enable communication among large and small businesses, residential customers and mobile users, the network 38 can consist of a combination of many individual sub-networks comprised of three main forms of interconnection. The trunk and main branches of the network 38 preferably employ fiber optic cable 40 as the preferred means of interconnection. Fiber optic cable 40 is used to connect between general purpose media processors 12 disposed as network servers 46 or large business installations 48 that are capable of coupling directly to the fiber optic link 40. For communications to small business and residential customers that may be incapable of directly coupling to the fiber optic cable 40, a general purpose media processor 12 can be used as an interface to other forms of network interconnection.

As shown in FIG. 2, alternate forms of interconnection consist of coaxial cable lines 42 and twisted pair wiring 44. Coaxial cable lines are currently in place throughout the U.S. and is typically employed to provide cable television services to residential homes. According to the preferred embodiment of the invention, general purpose media processors 12 can be installed at these residential locations 52. In contrast to the specialized processor approach, the general purpose media processor 12 provides enough bandwidth to allow for bi-directional communications to and from these residential locations 52.

Network servers 46 controlled by general purpose media processors 12 are also employed throughout the network 38. For example, the network servers 46 can be used to interface between the fiber optic network 40 and twisted pair wiring 44. Twisted pair wiring 44 is still employed for small businesses 50 and residential locations 52 that do not or cannot currently subscribe to coaxial cable or fiber optic network services. General purpose media processors 12 are also disposed at these small business locations 50 and non-cable residential locations 52. General purpose media processors 12 are also installed in wireless or mobile locations 52, which are coupled to the network 38 through agile digital radios (not shown). As shown in FIG. 2, network databases or other peripherals 56 can also be coupled to general purpose media processors 12 in the network 38.

10

The general purpose media processor 12 is operable at significantly high bandwidths in order to receive, process and transmit unified media data streams. Referring to FIG. 3, the respective frequencies for various types of media data streams are set forth against a bandwidth spectrum 60. The bandwidth spectrum 60 includes three component spectrums, all along the same range of frequencies, which represent the various frequency rates of digital media communications. Current computing bandwidth capabilities are also displayed. The telecommunications spectrum 62 shows the various frequency bands used for telecommunications transmission. For example, teletype terminals and modems operate in a range between approximately 64 kilobits/second to 16 kilobits/second. The ISDN telecommunication protocol operates at 64 kilobits/second. At the upper end of the telecommunications spectrum 62, T1 and T3 trunks operate at one megabit per second and 32 megabits per second, respectively. The SONET frequency range extends from approximately 128 megabits per second up to approximately 32 gigabits per second. Accordingly, in order to carry such broad band communications, the general purpose media processor 12 is capable of transferring information at rates into the gigabits per second range or higher.

A spectrum of typical media data streams is presented in the media spectrum 64 shown in FIG. 3. Voice and music transmissions are centered at frequencies of approximately 64 kilobits per second and one megabit per second, respectively. At the upper end of the media spectrum 64, video transmission takes place in a range from 128 megabits per second for high density television up to over 256 gigabits per second for movie applications. When using common video compression techniques, however, the video transmission spectrum can be shifted down to between 32 kilobits per second to 128 megabits per second as a result of the data compression. As described below, the processing required to achieve the data compression results in an increase in bandwidth requirements.

Current computing bandwidths are shown in the computing spectrum 66 of FIG. 3. Serial communications presently take place in a range between two kilobits per second up to 512 kilobits per second. The Ethernet network protocol operates at approximately 8 megabits per second. Current dynamic random access memory and other digital input/output peripherals operate between 32 megabits per second and 512 megabits per second. Presently available microprocessors are capable of operation in the low gigabits per second range. For example, the 386 Pentium microprocessor manufactured by Intel Corporation of Santa Clara, Calif. operates in the lower half of that range, and the Alpha microprocessor manufactured by Digital Equipment Corporation approaches the 16 gigabits per second range.

When video compression is employed, as expressed above, the associated processing overhead reduces the effective bandwidth of the particular processor. As a result, in order to handle compressed video, these processors must operate in the terahertz frequency range. The bandwidth spectrum 60 shown in FIG. 4 represents the effect of handling media data streams including compressed video. The computing spectrum 66 is skewed down to properly align the computing bandwidth requirements with the telecommunications spectrum 62 and the media spectrum 64. Accordingly, current processor technology is not sufficient to handle the transmission and processing associated with complex streams of multi-media data.

The current specialized processor approach to media processing is illustrated in the block diagram shown in FIG. 5. As shown in FIG. 5, special purpose processors are

5,809,321

11

coupled to a back plane 70, which is capable of transmitting instructions and data at the upper kilobits to lower gigabits per second range. In a typical configuration, an audio processor 76, video processor 78, graphics processor 80 and network processor 82 are all coupled to the back plane 70. Each of the audio, video, graphics and network processors 76-82 typically employ their own private or dedicated memories 84, which are only accessible to the specific processor and not accessible over the back plane 70. As described above, however, unless video data streams are constantly being processed, for example, the video processor 78 will sit idle for periods of time. The computing power of the dedicated video processor 78 is thus only available to handle video data streams and is not available to handle other media data streams that are directed to other dedicated processors. This, of course, is an inefficient use of the video processor 78 particularly in view of the overall processing capability of this multi-processor system.

The general purpose media processor 12, in contrast, handles a data stream of audio, video, graphics and network information all at the same time with the same processor. In order to handle the ever changing combination of data types, the general purpose media processor 12 is dynamically partitionable to allocate the appropriate amount of processing for each combination of media in a unified media data stream. A block diagram of two preferred general purpose media processor system configurations is shown in FIG. 6. Referring to FIG. 6(a), a general purpose media processor 12 is coupled to a high-speed back plane 90. The presently preferred back plane 90 is capable of operation at 30 gigabits per second. As those skilled in the art will appreciate, back planes 90 that are capable of operation at 400 gigabits per second or greater bandwidth are envisioned within the spirit and scope of the invention. Multiple memory devices 92 are also coupled to the back plane 90, which are accessible by the general purpose media processor 12. Input/output devices 94 are coupled to the back plane 90 through a dual-ported memory 92. The configuration of the input/output devices 94 on one end of the dual-ported memory 92 allows the sharing of these memory devices 92 throughout a network 38 of general purpose media processors 12.

Alternatively, FIG. 6(b) shows a presently preferred integrated general purpose media processor 12. The integrated processor includes on-board memory and I/O 86. The on-board memory is preferably of sufficient size to optimize throughput, and can comprise a cache and/or buffer memory or the like. The integrated media processor 12 also connects to external memory 88, which is preferably larger than the on-board memory 86 and forms the system main memory. Execution Unit

One presently preferred embodiment of an integrated general purpose media processor 12 is shown in FIG. 7. The core of the integrated general purpose media processor 12 comprises an execution unit 100. Three main elements or subsections are included in the execution unit 100. A multiple precision arithmetic/logic unit ("ALU") 102 performs all logical and simple arithmetic operations on incoming media data streams. Such operations consist of calculate and control operations such as Boolean functions, as well as addition, subtraction, multiplication and division. These operations are performed on single or unified media data streams transmitted to and from the multiple precision ALU 102 over a data bus or data path 108. Preferably the data path 108 is 128 bits wide, although those skilled in the art will appreciate that the data path 108 can take on any width or size without departing from the spirit and scope of the invention. The wider the data path 108 the more unified

12

media data can be processed in parallel by the general purpose media processor 12.

Coupled to the multi-precision ALU 102 via the data path 108, and also an element of the execution unit 100, is a programmable switch 104. The programmable switch 104 performs data handling operations on single or unified media data streams transmitted over the data path 108. Examples of such data handling operations include deals, shuffles, shifts, expands, compresses, swizzles, permutes and reverses, although other data handling operations are contemplated. These operations can be performed on single bits or bit fields consisting of two or more bits up to the entire width of the data path 108. Thus, single bits or bit fields of various sizes can be manipulated through programmable operation of the switch 104.

Examples of the presently preferred data manipulation operations performed by the general purpose media processor 12 are shown in FIG. 8. A group expand operation is visually illustrated in FIG. 8(a). According to the group expand operation, a sequential field of bits 270 can be divided into constituent sub-fields 272a-272d for insertion into a larger field array 274. The reverse of the group expand operation is a group compress or extract operation. A visual illustration of the group compress or extract operation is shown in FIG. 8(b). As shown, separate sub-fields 272a-272d from a larger bit field 274 can be combined to form a contiguous or sequential field of bits 270.

Referring to FIGS. 8(c)-8(e), group deal, shuffle, swizzle and permute operations performed by the programmable switch 104 are also illustrated. The operations performed by these instructions are readily understood from a review of the drawings. The group manipulation operations illustrated in FIGS. 8(a)-8(e) comprise the presently contemplated data manipulation operations for the general purpose media processor 12. As those skilled in the art will appreciate, either a subset of these operations or additional data manipulation operations can be incorporated in other alternate embodiments of the general purpose media processor 12 without departing from the spirit and scope of the invention.

Referring again to FIG. 7, higher level mathematical operations than those performed by the multi-precision ALU 102 are performed in the general purpose media processor 12 through an extended math element 106. The extended math element 106 is coupled to the data path 108 and also comprises part of the execution unit 100. The extended math element 106 performs the complex arithmetic operations necessary for video data compression and similarly intensive mathematical operations. One presently preferred example of an extended math operation comprises a Galois field operation. Other examples of extended mathematical functions performed by the extended math element 106 include CRC generation and checking, Reed-Solomon code generation and checking, and spread-spectrum encoding and decoding. As those skilled in the art appreciate, additional mathematical operations are possible and contemplated.

According to the preferred embodiment of the integrated general purpose media processor 12, a register file 110 is provided in addition to the execution unit 100 to process media data. The register file 110 stores and transmits data streams to and from the execution unit 100 via the data path 108. Rather than employing a complex set of specific or dedicated registers, the general purpose media processor 12 preferably includes 64 general purpose registers in the register file 110 along with one program counter (not shown). The 64 general purpose registers contained in the register file 110 are all available to the user/programmer, and comprise a portion of the user state of the general purpose

5,809,321

13

media processor 12. The general purpose registers are preferably capable of storing any form of data. Each register within the register file 110 is coupled to the data path 108 and is accessible to the execution unit 100 in the same manner. Thus, the user can employ a general purpose register according to the specific needs of a particular program or unique application. As those skilled in the art will appreciate, the register file 110 can also comprise a plurality of register files 110 configured in parallel in order to support parallel multi-threaded processing.

Instruction Set and User Programming

Control or manipulation of data processed by the general purpose media processor 12 is achieved by selected instructions programmed by the user. Those skilled in the art will appreciate that a great number of programs are possible through various sequences of instructions. Particular programs can be developed for each unique implementation of the general purpose media processor 12. A detailed discussion of such specific programs is therefore beyond the scope of this description.

One presently preferred instruction set for the general purpose media processor 12 is included in the Microfiche Appendix, the contents of which are hereby incorporated herein by reference. A list of the presently preferred major operation codes for the general purpose media processor 12 appears below in Table I.

14

instruction. As many as 255 separate operations are contemplated for the preferred embodiment of the general purpose media processor 12. As shown in Table I, however, not all of the operation codes are presently implemented. As those skilled in the art will appreciate, alternate schemes for organizing the operation codes, as well as additional operation codes for the general purpose media processor 12, are possible.

The instructions provided in the instruction set for the general purpose media processor 12 control the transfer, processing and manipulation of data streams between the register file 110 and the execution unit 100. The presently preferred width of the instruction path 112 is 32-bits wide, organized as four eight-bit bytes ("quadlets"). Those skilled in the art will appreciate, however, that the instruction path 112 can take on any width without departing from the spirit and scope of the invention. Preferably, each instruction within the instruction set is stored or organized in memory on four-byte boundaries. The presently preferred format for instructions is shown in FIG. 9(a).

As shown in FIG. 9(a), each of the presently preferred instruction formats for the general purpose media processor 12 includes a field 280 for the major operation code number shown in Table I. Based on the type of operation performed, the remaining bits can provide additional operands according to the type of addressing employed with the operation.

TABLE I

MAJOR OPERATION CODES							
MAJOR	0	32	64	96	128	160	192
JOR							
major operation code field values							
0	ERES	GSHUFFLEI	FMULADD16	GMULADD1	LU16LAI	SAAS54LAI	EADDIO
1	ESHUFFLEI4MUX	GSHUFFLEI4MUX	FMULADD32	GMULADD2	LU16BAI	SAAS64BAI	EADDIUO
2		GSELECT8	FMULADD64	GMULADD4	LU16LI	SCAS64LAI	ESETIL
3	EMDEPI	GMDEPI		GMULADD8	LU16BI	SCAS64BAI	ESETIGE
4	EMUX	GMUX	FMULSUB16	GMULADD16	LU32LAI	SMAS64LAI	ESETIE
5	FSMUX	G8MUX	FMULSUB32	GMULADD32	LU32BAI	SMAS64BAI	ESETINE
6	EGFMUL64	GGFMUL8	FMULSUB64	GMULADD64	LU32LI	SMUX64LAI	ESETIUL
7	ETRANSPOSE8MUX	GTRANSPOSE8MUX		GEXTRACT128	LU32BI	SMUX64BAI	ESETIUGE
8					L16LAI	S16LAI	ESUBIO
9	ESWIZZLE	GSWIZZLE		GUMULADD2	L16BAI	S16BAI	ESUBIUO
10		GSWIZZLECOPY		GUMULADD4	L16LI	S16LI	ESUBIL
11		GSWIZZLESWAP		GUMULADD8	L16BI	S16BI	ESUBIGE
12	EDEPI	GDEPI	F.16	GUMULADD16	L32LAI	S32LAI	ESUBIE
13	EUDEPI	GUDEPI	F.32	GUMULADD32	L32BAI	S32BAI	ESUBINE
14	EWTHI	GWTHI	F.64	GUMULADD64	L32LI	S32LI	ESUBIUL
15	EUWTHI	GUWTHI		GUEXTRACT128	L32BI	S32BI	ESUBIUGE
16			GFMULADD16	GEXTTRACT1	L64LAI	S64LAI	EADDI
17			GFMULADD32	GEXTTRACT16	L64BAI	S64BAI	EXORI
18			GFMULADD64	GEXTTRACT32	L64LI	S64LI	EORI
19			GFMULADD128	GEXTTRACT64	L64BI	S64BI	EANDI
20			GFMULSUB16	GEXTTRACT	L128LAI	S128LAI	ESUBI
21			GFMULSUB32	L.54	L128BAI	S128BAI	BNE
22			GFMULSUB64	G.EXTTRACT	L128LI	S128LI	ENORI
23			GFMULSUB128	L.128	L128BI	S128BI	ENANDI
24				G.1	LU8I	S8I	BGATEI
25				G.2	LU8I		
26				G.4			
27				G.8			
28		FCOPYI	GF.16	G.16			FCOPYI
29			GF.32	G.32			BLINKI
30			GF.64	G.64			
31		EMINOR	GF.128	G.128	L.MINOR	S.MINOR	E.MINOR

As shown in Table I, the major operation codes are grouped according to the function performed by the operations. The operations are thus arranged and listed above according to the presently preferred operation code number for each

For example, the remainder of the 32-bit instruction field can comprise an immediate operand ("imm"), or operands stored in any of the general registers ("ra", "rb", "rc", and "rd"). In

5,809,321

15

addition, minor operation codes 282 can also be included among the operands of certain 32-bit instruction formats.

The presently preferred embodiment of the general purpose media processor 12 includes a limited instruction set similar to those seen in Reduced Instruction Set Computer ("RISC") systems. The preferred instruction set for the general purpose media processor 12 shown in Table I includes operations which implement load, store, synchronize, branch and gateway functions. These five groups of operations can be visually represented as two general classes of related operations. The branch and gateway operations perform related functions on media data streams and are thus visually represented as block 114 in FIG. 7. Similarly, the load, store and synchronize operations are grouped together in block 116 and perform similar operations on the media data streams. (Blocks 114 and 116 only represent the above classification of these operations and their function in the processing of media data streams, and do not indicate any specific underlying electronic connections.) A more detailed discussion of these operations, and the functionality of the general purpose media processor 12, appears in the Microfiche Appendix.

The four-byte structure of instructions for the general purpose media processor 12 is preferably independent of the byte ordering used for any data structures. Nevertheless, the gateway instructions are specifically defined as 16-byte structures containing a code address used to securely invoke a procedure at a higher privilege level. Gateways are preferably marked by protection information specified in the translation lookaside buffer 148 in the memory management unit 122. Gateways are thus preferably aligned on 16-byte boundaries in the external memory. In addition to the general purpose registers and program counter, a privilege level register is provided within the register file 110 that contains the privilege level of the currently executing instruction.

The instruction set preferably includes load and store instructions that move data between memory and the register file 110, branch instructions to compare the content of registers and transfer control, and arithmetic operations to perform computations on the contents of registers. Swap instructions provide multi-thread and multi-processor synchronization. These operations are preferably indivisible and include such instructions as add-and-swap, compare-and-swap, and multiplex-and-swap instructions. The fixed-point compare-and-branch instructions within the instruction set shown in Table I provide the necessary arithmetic tests for equality and inequality of signed and unsigned fixed-point values. The branch through gateway instruction provides a secure means to access code at a higher privileged level in a form similar to a high level language procedure call generally known in the art.

The general purpose media processor 12 also preferably supports floating-point compare-and-branch instructions. The arithmetic operations, which are supported in hardware, include floating-point addition, subtraction, multiplication, division and square root. The general purpose media processor 12 preferably supports other floating-point operations defined by the ANSI-IEEE floating-point standard through the use of software libraries. A floating point value can preferably be 16, 32, 64 or 128-bits wide. Examples of the presenting preferred floating-point data sizes are illustrated in FIG. 9(b).

The general purpose media processor 12 preferably supports virtual memory addressing and virtual machine operation through a memory management unit 122. Referring to FIG. 10(a), one presently preferred embodiment of the memory management unit 122 is shown. The memory

16

management unit 122 preferably translates global virtual addresses into physical addresses by software programmable routines augmented by a hardware translation lookaside buffer ("TLB") 148. A facility for local virtual address translation 164 is also preferably provided. As those skilled in the art will appreciate, the memory management unit 122 includes a data cache 166 and a tag cache 168 that store data and tags associated with memory sections for each entry in the TLB 148.

A block diagram of one preferred embodiment of the TLB 148 is shown in FIG. 10(b). The TLB 148 receives a virtual address 230 as its input. For each entry in the TLB 148, the virtual address 230 is logically AND-ed with a mask 232. The output of each respective AND gate 234 is compared via a comparator 236 with each entry in the TLB 148. If a match is detected, an output from the comparator 236 is used to gate data 240 through a transceiver 238. As those skilled in the art will appreciate, a match indicates the entry of the corresponding physical address within the contents of the TLB 148 and no external memory or I/O access is required. The data 240 for the data cache 166 (FIG. 10(a)) is then combined with the remaining lower bits of the virtual address 230 through an exclusive-OR gate 242. The resultant combination is the physical address 244 output from the TLB 148. If a match is not detected between the logical address and the contents of the tag cache 168, the memory management unit 122 an external memory or I/O access is necessary to retrieve the relevant portion of memory and update the contents of the TLB 148 accordingly.

Using generally known memory management techniques, the memory management unit 122 ensures that instructions (and data) are properly retrieved from external memory (or other sources) over an external input/output bus 126 (see FIG. 7). As described in more detail below, a high bandwidth interface 124 is coupled to the external input/output bus 126 to communicate instructions (and media data streams) to the general purpose media processor 12. The presently preferred physical address width for the general purpose media processor 12 is eight bytes (64-bits). In addition, the memory management unit 122 preferably provides match bits (not shown) that allow large memory regions to be assigned a single TLB entry allowing for fine grain memory management of large memory sections. The memory management unit 122 also preferably includes a priority bit (not shown) that allows for preferential queuing of memory areas according to respective levels of priority. Other memory management operations generally known in the art are also performed by the memory management unit 122.

Referring again to FIG. 7, instructions received by the general purpose media processor 12 are stored in a combined instruction buffer/cache 118. The instruction buffer/cache 118 is dynamically subdivided to store the largest sequence of instructions capable of execution by the execution unit 100 without the necessity of accessing external memory. In a preferred embodiment of the invention, instruction buffer space is allocated to the smallest and most frequently executed blocks of media instructions. The instruction buffer thus helps maintain the high bandwidth capacity of the general purpose media processor 12 by sustaining the number of instructions executed per second at or near peak operation. That portion of the instruction buffer/cache 118 not used as a buffer is, therefore, available to be used as cache memory. The instruction buffer/cache 118 is coupled to the instruction path 112 and is preferably 32 kilobytes in size.

A data buffer/cache 120 is also provided to store data transmitted and received to and from the execution unit 100

5,809,321

17

and register file 110. The data buffer/cache 120 is also dynamically subdivided in a manner similar to that of the instruction buffer/cache 118. The buffer portion of the data buffer/cache 120 is optimized to store a set size of unified media data capable of execution without the necessity of accessing external memory. In a preferred embodiment of the invention, data buffer space is allocated to the smallest and most frequently accessed working sets of media data. Like the instruction buffer, the data buffer thus maintains peak bandwidth of the general purpose media processor 12. The data buffer/cache 120 is coupled to the data path 108 and is preferably also 32 kilobytes in size.

The preferred embodiment of the general purpose media processor 12 includes a pipelined instruction pre-fetch structure. Although pipelined operation is supported, the general purpose media processor 12 also allows for non-pipelined operations to execute without any operational penalty. One preferred pipeline structure for the general purpose media processor 12 comprises a "super-string" pipeline shown in FIG. 11. A super-string pipeline is designed to fetch and execute several instructions in each clock cycle. The instructions available for the general purpose media processor 12 can be broken down into five basic steps of operation. These steps include a register-to-register address calculation, a memory load, a register-to-register data calculation, a memory store and a branch operation. According to the super-string pipeline organization of the general purpose media processor 12, one instruction from each of these five types may be issued in each clock cycle. The presently preferred ordering of these operations are as listed above where each of the five steps are assigned letters "A," "L," "E," "S" and "B" (see FIG. 11).

According to the super-string pipelining technique, each of the instructions are serially dependent, as shown in FIG. 11, and the general purpose media processor 12 has the ability to issue a string of dependent instructions in a single clock cycle. These instructions shown in FIG. 11 can take from two to five cycles of latency to execute, and a branch prediction mechanism is preferably used to keep up the pipeline filled (described below). Instructions can be encoded in unit categories such as address, load, store/sync, fixed, float and branch to allow for easy decoding. A similar scheme is employed to pre-fetch data for the general purpose media processor 12.

As those skilled in the art will appreciate, the super-string pipeline can be implemented in a multi-threaded environment. In such an implementation, the number of threads is preferably relatively prime with respect to functional unit rates so that functional units can be scheduled in a non-interfering fashion between each thread.

In another more preferred embodiment, a "super-spring" pipelining scheme is employed with the general purpose media processor 12. The super-spring pipeline technique breaks the super-string pipeline shown in FIG. 11 into two sections that are coupled via a memory buffer (not shown). A visual representation of the super-spring pipeline technique is shown in FIG. 12. The front of the pipeline 204, in which address calculation (A), memory load (L), and branch (B) operations are handled, is decoupled from the back of the pipeline 206, in which data calculation (E) and memory store (S) operations are handled. The decoupling is accomplished through the memory buffer (not shown), which is preferably organized in a first-in-first-out ("FIFO") fast/dense structure. (The memory buffer is functionally represented as a spring in FIG. 12.)

As indicated in Table I above, the general purpose media processor 12 does not include delayed branch instructions,

18

and so relies upon branch or fetch prediction techniques to keep the pipeline full in program flows around unconditional and conditional branch instructions. Many such techniques are generally known in the art. Examples of some presently preferred techniques include the use of group compare and set, and multiplex operations to eliminate unpredictable branches; the use of short forward branches, which cause pipeline neutralization; and where branch and link predicts the return address in a one or more entry stack. In addition, the specialized gateway instructions included in the general purpose media processor 12 allow for branches to and from protected virtual memory space. The gateway instructions, therefore, allow an efficient means to transfer between various levels of privilege.

As described above, two basic forms of media data are processed by the general purpose media processor 12, as shown in FIG. 7. These data streams generally comprise Nyquist sampled I/O 128, and standard memory and I/O 130. As shown in FIG. 7, audio 132, video 134, radio 136, network 138, tape 140 and disc 142 data streams comprise some examples of digitally sampled I/O 128. As those skilled in the art will appreciate, other forms of digitally sampled I/O are contemplated for processing by the general purpose media processor 12 without departing from the spirit and scope of the invention. Standard memory and I/O 130 comprises data received and transmitted to and from general digital peripheral devices used in the design of most computer systems. As shown in FIG. 7, some examples of such devices include dynamic random access memory ("DRAM") 146, or any data received over the PCI bus 144 generally known in the art. Other forms of standard memory and I/O sources are also contemplated. The various fixed-point data sizes preferred for the general purpose media processor 12 are illustrated in FIG. 9(c).

External Interface

As mentioned above, the general purpose media processor 12 includes a high bandwidth interface 124 to communicate with external memory and input/output sources. As part of the high bandwidth interface 124, the general purpose media processor 12 integrates several fast communication channels 156 (FIG. 13) to communicate externally. These fast communication channels 156 preferably couple to external caches 150, which serve as a buffer to memory interfaces 152 coupled to standard memory 154. The caches 150 preferably comprise synchronous static random access memory ("SRAM"), each of which are sixty-four kilobytes in size; and the standard memories 154 comprise DRAM's. The memory interfaces 152 transmit data between the caches 150 and the standard memories 154. The standard memories 154 together form the main external memory for the general purpose media processor 12. The cache 150, memory interface 152, standard memory 154 and input/output channel 156 therefore make up a single external memory unit 158 for the general purpose media processor 12.

According to the presently preferred embodiment of the invention, the memory interface protocol embeds read and write operations to a single memory space into packets containing command, address, data and acknowledgment information. The packets preferably include check codes that will detect single-bit transmission errors and some multiple-bit errors. As many as eight operations may be in progress at a time in each external memory unit 158. As shown in FIG. 13, up to four external memory units 158 may be cascaded together to expand the memory available to the general purpose media processor 12, and to improve the bandwidth of the external memory. Through such cascaded

5,809,321

19

memory units 158, the memory interface 152 provides for the direct connection of multiple banks of standard memory 154 to maintain operation of the general purpose media processor 12 at sustained peak bandwidths.

According to one embodiment shown in FIG. 13, up to four standard memory devices 154 can be coupled to each memory interface 152. Each standard memory 154 thus includes as many as four banks of DRAM, each of which is preferably sixteen bits wide. The standard memories 154 are connected in parallel to the memory interface 152 forming a 72-bit wide data bus 160, where 64 bits are preferably provided for data transfer and eight bits are provided for error correction. In addition to the data bus 160, an address/control bus 162 is coupled between the memory interface 152 and each standard memory 154. The address/control bus 162 preferably comprises at least twelve address lines (4 kilobits \times 15 memory size) and four control lines as shown in FIG. 13. An alternate manner for coupling the DRAM's to the memory interface 152 is illustrated in FIG. 14. As shown in FIG. 14, two banks of four DRAM single in-line memory modules are coupled in parallel to the memory interface 152. The memory interface 152 also supports interleaving to enhance bandwidth, and page mode accesses to improve latency for localized addressing.

Using standard DRAM components, the external memory units 158 achieve bandwidths of approximately two gigabits/second with the standard memories 154. When four such external memory units 158 are coupled via the communication channel 156, therefore, the total bandwidth of the external main memory system increases to one gigabyte/second. As discussed further below, in implementations with two or eight communication channels 156, the aggregate bandwidth increases to two and eight gigabytes/second, respectively.

A more detailed depiction of the communication channel 156 circuitry appears in FIG. 15. According to the preferred embodiment of the invention, each communication channel 156 comprises two unidirectional, byte-wide, differential, packet-oriented data channels 156a, 156b (see FIG. 13). As explained above, where memory units 158 are cascaded together in series, the output of one memory unit 158 is connected to the input of another memory unit 158. The two unidirectional channels are thus connected through the memory units 158 forming a loop structure and make up a single bi-directional memory interface channel.

Referring to FIG. 15, each communication channel 156 is preferably eight bits wide, and each bit is transmitted differentially. For example, output transceiver 170 for bit D_{0out} transmits both D_0 and $\overline{D_0}$ signals over the communication channel 156. Additional transceivers are similarly provided for the remaining bits in the channel 156. (The transceiver 176 for bit D_{7out} and associated differential lines 178, 180 are shown in FIG. 15.) A CLK_{out} transceiver 182 is also provided to generate differential clock outputs 184, 186 over the channel 156. To complete the link between memory units 158, input transceivers 188–192 are provided in each memory unit 158 for each of the differential bits and clock signals transmitted over the communication channel 156. These input signals 172, 174, 178, 180, 184, 186 are preferably transmitted through input buffers 194–198 to other parts of the memory unit 158 (described above).

Each memory unit 158 also includes a skew calibrator 200 and phase locked loop ("PLL") 202. The skew calibrator 200 is used to control skew in signals output to the communication channel 156. Preferably, digital skew fields are employed, which include set numbers of delay stages to be inserted in the output path of the communication channel

20

156. Setting these fields, and the corresponding analog skew fields, permits a fine level of control over the relative skew between output channel signals.

The PLL 202 recovers the clock signal on either side of the communication channel 156 and is thus provided to remove clock jitter. The clock signals 184, 186 preferably comprise a single phase, constant rate clock signal. The clock signals 184, 186 thus contain alternating zero and one values transmitted with the same timing as the data signals 172, 174, 178, 180. The clock signal frequency is, therefore, one-half the byte data rate. The communication channel 156 preferably operates at constant frequency and contains no auxiliary control, handshaking or flow control information.

Each external memory unit 158 preferably defines two functional regions: a memory region, implemented by the cache 150 backed by standard memory 154 (see FIG. 13), and a configuration region, implemented by registers (not shown). Both regions are accessed by separate interfaces; the communication channel 156 is used to access the memory region, and a serial interface (described below) is used to access the configuration region. In the memory region, the caches 150 are preferably write-back (write-in) single-set (direct-map) caches for data originally contained in standard memory 154. All accesses to memory space should maintain consistency between the contents of the cache 150 and the contents of the standard memory 154. The configuration region registers provide the mechanism to detect and adjust skew in the communication channel 156. Software is preferably employed to adaptively adjust the skew in the channel 156 through digital skew fields, as explained above. The serial interface thus is used to configure the external memory units 158, set diagnostic modes and read diagnostic information, and to enable the use of a high-speed tester (not shown).

One presently preferred embodiment of the invention employs two byte-wide packet communication channels 156 (FIG. 16(a)). In order to further increase the bandwidth of the general purpose media processor 12, up to sixteen byte-wide packet communication channels 156 can be employed. Referring to FIG. 16(b), twelve communication channels, comprising eight memory channels 210, a ninth channel for parallel processing 212 (described below), and three input/output ("I/O") channels 214, are shown. Each of the communication channels 210–214 preferably employs the cascade configuration of four channel interface devices 216. (Each channel interface device 216 coupled to the memory channels 210 corresponds to the external memory unit 158 shown in FIG. 13.) Through each of the twelve communication channels shown in FIG. 16(b), the general purpose media processor 12 can request or issue read or write transactions. When not interleaved, the twelve channels provide a single contiguous memory space for each channel interface device 216.

Alternatively, memory accesses may be interleaved in order to provide for continuous access to the external memory system at the maximum bandwidth for the DRAM memories. In an interleaved configuration, at any point in time some memory devices will be engaged in row pre-charge, while others may be driving or receiving data, or receiving row or column addresses. The memory interface 152 (FIG. 13) thus preferably maps between a contiguous address space and each of the separate address spaces made available within each external memory unit 158. For maximum performance, therefore, the memory interface is interleaved so that references to adjacent addresses are handled by different memory devices. Moreover, in the preferred embodiment, additional memory operations may be

5,809,321

21

requested before the corresponding DRAM bank is available. In an interleaved approach, these operations are placed in a queue until they can be processed. According to the preferred embodiment, memory writes have lower priority than memory reads, unless an attempt is made to read an address that is queued for a write operation. As those skilled in the art will appreciate, the depth of the memory write queue is dictated by the specific implementation.

Although up to four external memory units 158 are preferably cascaded to form effectively larger memories, some amount of latency may be introduced by the cascade. Packets of data transmitted over the communication channel 156 are uniquely addressed to a particular channel interface device 216. A packet received at a particular device, which specifies another module address, is automatically passed to the correct channel interface device 216. Unless the module address matches a particular device 216, that packet simply passes from the input to the output of the interface device 216. This mechanism divides the serial interconnection of interface devices 216 into strings, which function as a single larger memory or peripheral, but with possibly longer response latency.

In addition to the memory channels 210, the general purpose media processor 12 provides several communication channels 214 for communication with external input/output devices. Referring to FIG. 16(b), three input/output channels 214 having SRAM buffered memory (see FIG. 13) provide an interface to external standard I/O devices (not shown). Like the eight memory channels 210, the three I/O channels 214 are byte-wide input/output channels intended to operate at rates of at least one gigahertz. The three I/O channels 214 also operate as a packet communication link to synchronous SRAM memory 208 within the channel interface device 216. A controller 226 within the channel interface device 216 completes the interface to the I/O devices.

The three I/O channels 214 preferably function in like manner to the memory channels 210 described above. The interface protocol for the three I/O channels 214 divides read and write operations to a single memory space into packets containing command, address, data and acknowledgment information. The packets also include a check code that will detect single-bit transmission errors and some multiple-bit errors. According to the preferred embodiment of the invention, as many as eight operations may progress in each interface device 216 at a time. As shown in FIG. 16(b), up to four channel interface devices 216 can be cascaded together to expand the bandwidth in the three I/O channels 214. A bit-serial interface (not shown) is also provided to each of the channel interface devices 216 to allow access to configuration, diagnostic and tester information at standard TTL signal levels at a more moderate data rate. (A more detailed description of the serial interface is provided below).

Like the memory channels 210, each I/O channel 214 includes nine signals—one clock signal and eight data signals. Differential voltage levels are preferably employed for each signal. Each channel interface device 216 is preferably terminated in a nominal 50 ohm impedance to ground. This impedance applies for both inputs and outputs to the communication channel 156. A programmable termination impedance is preferred.

Interface Communication

According to one presently preferred embodiment of the invention, the channel interface devices 216 can operate as either master devices or slave devices. A master device is capable of generating a request on the communication channel 156 and receiving responses from the communi-

22

cation channel 156. Slave devices are capable of receiving requests and generating responses, over the communication channel 156. A master device is preferably capable of generating a constant frequency clock signal and accepting signals at the same clock frequency over the communication channel 156. A slave device, therefore, should operate at the same clock rate as the communication channel 156, and generate no more than a specified amount of variation in output clock phase relative to input clock phase. The master device, however, can accept an arbitrary input clock phase and tolerates a specified amount of variation in clock phase over operating conditions.

Packets of information sent over the communication channel 156 preferably contain control commands, such as read or write operations, along with addresses and associated data. Other commands are provided to indicate error conditions and responses to the above commands. When the communication channel 156 is idle, such as during initialization and between transmitted packets, an idle packet, consisting of an all-zero byte and an all-one byte is transmitted through the communication channel 156. Each non-idle packet consists of two bytes or a multiple of two bytes, and begins with a byte having a value other than all zeros. All packets transmitted over the communication channel 156 also begin during a clock period in which the clock signal is zero, and all packets preferably end during a clock period in which the clock signal is one. A depiction of the preferred packet protocol format for transmission over the communication channel 156 appears in FIG. 17.

The general form of each packet is an array of bytes preferably without a specific byte ordering. The first byte contains a module address 250 ("ma") in the high order two bits; a packet identifier, usually a command 252 ("com"), in the next three bit positions; and a link identification number 254 ("lid") in the last three bit positions. The interpretation of the remaining bytes of a packet depend upon the contents of the packet identifier. The length of each packet is preferably implied by the command specified in the initial byte of the packet. A check byte is provided and computed as odd bit-wise parity with a leftward circular rotation after accumulating each byte. This technique provides detection of all single-bit and some multiple-bit errors, but no correction is provided.

The modular address 250 field of each packet is preferably a two-bit field and allows for as many as four slave devices to be operated from a single communication channel 156. Module address values can be assigned in one of two fashions: either dynamically assigned through a configuration register (not shown), or assigned via static/geometric configuration pins. Dynamic assignment through a configuration register is the presently preferred method for assigning module address values.

The link identification number 254 field is preferably 3-bits wide and provides the opportunity for master devices to initiate as many as eight independent operations at any one time to each slave device. Each outstanding operation requires a distinct link identification number, but no ordering of operations should be implied by the value of the link identification field. Thus, there is preferably no requirement for link identification values 254 to be sequentially assigned either in requests or responses.

The receipt of packets over the communication channel 156 that do not conform to the channel protocol preferably generates an error condition. As those skilled in the art will appreciate, the level or degrees to which a specific implementation detects errors is defined by the user. In one presently preferred embodiment of the invention, all errors

5,809,321

23

are detected, and the following protocol is employed for handling errors. For each error detected, the channel interface device 216 causes a response explicitly indicating the error condition. Channel interface devices 216 reporting an invalid packet will then suppress the receipt of additional packets until the error is cleared. The transmitted packet is otherwise ignored. However, even though the erroneous packet is ignored, the channel interface devices 216 preferably continue to process valid packets that have already been received and generate responses thereto. An identification of the presently preferred commands 252 to be used over the communication channel 156 are listed in FIG. 17.

In the master/slave preferred embodiment, the channel interface devices 216 forward packets that are intended for other devices connected to the communication channel 156, as described above. In slave devices, forwarding is performed based on the module address 250 field of the packet. Packets which contain a module address 250 other than that of the current device are forwarded on to the next device. All non-idle packets are thus forwarded including error packets. In master devices, forwarding is performed based on the link identifier number 254 of the packet. Packets that contain link identifier numbers 254 not generated by the specific channel interface device 216 are forwarded. In order to reduce transmission latency, a packet buffer may be provided. As those skilled in the art appreciate, the suitable size for the packet buffer depends on the amount of latency tolerable in a particular implementation.

A variety of master/slave ring configurations are possible using the high bandwidth interface 124 of the invention. Five ring configurations are currently preferred: single-master, dual-master, multiple-master, single-slave and multiple-master/multiple-slave. The simplest ring configuration contains a single non-forwarding master device and a single non-forwarding slave device. No forwarding is required for either device in this configuration as packets are sent directly to the recipient. A single-master ring, however, may contain a cascade of up to four slave devices (see FIGS. 13, 16). In the single-master ring configuration, each slave device is configured to a distinct module address, and each slave device forwards packets that contain module address fields unequal to their own. As discussed above, a single-master ring provides a larger memory or I/O capacity than a master-slave pair, but also introduces a potentially longer response latency. In the single-master ring, each slave device may have as many as eight transactions outstanding at any time, as described above.

The remaining combinations share many of the above basic attributes. In a dual-master pair, each master device may initiate read and write operations addressed to the other, and each may have up to eight such transactions outstanding. No forwarding is required for either device because packets are sent directly to the recipient. A multiple-master ring may contain multiple master devices and a single slave device. In this configuration, the slave device need not forward packets as all input packets are designated for the single slave device. A multiple-master ring may contain multiple master devices and as many as four slave devices. Each slave device may have up to eight transactions outstanding, and each master device may use some of those transactions. In a preferred embodiment, a master also has the capability to detect a time-out condition or when a response to a request packet is not received. Further aspects of interprocessor communications and configurations are discussed below in connection with FIG. 18.

Serial Bus

In one preferred embodiment of the invention, the general purpose media processor 12 includes a serial bus (not

24

shown). The serial bus is designed to provide bootstrap resources, configuration, and diagnostic support to the general purpose media processor 12. The serial bus preferably employs two signals, both at TTL levels, for direct communication among many devices. In the preferred embodiment, the first signal is a continuously running clock, and the second signal is an open-collector bi-directional data signal. Four additional signals provide geographic addresses for each device coupled to the serial bus. A gateway protocol, and optional configurable addressing, each provide a means to extend the serial bus to other buses and devices. Although the serial bus is designed for implementation in a system having a general purpose media processor 12, as those skilled in the art will appreciate, the serial bus is applicable to other systems as well.

Because the serial bus is preferably used for the initial bootstrap program load of the general purpose media processor 12, the bootstrap ROM is coupled to the serial bus. As a result, the serial bus needs to be operational for the first instruction fetch. The serial bus protocol is therefore devised so that no transactions are required for initial bus configuration or bus address assignment.

According to the preferred embodiment, the clock signal comprises a continuously running clock signal at a minimum of 20 megahertz. The amount of skew, if any, in the clock signal between any two serial bus devices should be limited to be less than the skew on the data signal. Preferably, the serial data signal is a non-inverted open collector bi-directional data signal. TTL levels are preferred for communication on the serial bus, and several termination networks may be employed for the serial data signal. A simple preferred termination network employs a resistive pull-up of 220 ohms to 3.3 volts above V_{cc} . An alternate embodiment employs a more complex termination network such as a termination network including diodes or the "Forced Perfect Termination" network proposed for the SCSI-2 standard, which may be advantageous for larger configurations.

The geographic addressing employed in the serial bus is provided to insure that each device is addressable with a number that is unique among all devices on the bus and which also preferably reflects the physical location of the device. Thus, the address of each device remains the same each time the system is operated. In one preferred embodiment, the geographic address is composed of four bits, thus allowing for up to 16 devices. In order to extend the geographic addressing to more than 16 devices, additional signals may be employed such as a buffered copy of the clock signal or an inverted copy of the clock signal (or both).

The serial bus preferably incorporates both a bit level and packet protocol. The bit level protocol allows any device to transmit one bit of information on the bus, which is received by all devices on the bus at the same time. Each transmitted bit begins at the rising edge of the clock signal and ends at the next rising edge. The transmitted bit value is sampled at the next rising edge of the clock signal. According to one preferred embodiment where the serial data signal is an open collector signal, the transmission of a zero bit value on the bus is achieved by driving the serial data signal to a logical low value. In this embodiment, the transmission of a one bit value is achieved by releasing the serial data signal to obtain a logical high value. If more than one device attempts to transmit a value on the same clock, the resulting value is a zero if any device transmits a zero value, and one if all devices transmit a one value. This provides a "wired-AND" collision mechanism, as those skilled in the art will appreciate. If two or more devices transmit the same value on the

5,809,321

25

same clock cycle, however, no device can detect the occurrence of a collision. In such cases, the transaction, which may occur frequently in some implementations, preferably proceeds as described below.

The packet protocol employed with the serial bus uses the bit level protocol to transmit information in units of eight bits or multiples of eight bits. Each packet transmission preferably begins with a start bit in which the serial data signal has a zero (driven) value. After transmitting the eight data bits, a parity bit is transmitted. The transmission continues with additional data. A single one (released) bit is transmitted immediately following the least significant bit of each byte signaling the end of the byte.

On the cycle following the transmission of the parity bit, any device may demand a delay of two cycles to process the data received. The two cycle delay is initiated by driving the serial data signal (to a zero value) and releasing the serial data signal on the next cycle. Before releasing the serial data signal, however, it is preferable to insure that the signal is not being driven by any other device. Further delays are available by repeating this pattern.

In order to avoid collisions, a device is not permitted to start a transmission over the serial bus unless there are no currently executing transactions. To resolve collisions that may occur if two devices begin transmission on the same cycle, each transmitting device should preferably monitor the bus during the transmission of one (released) bits. If any of the bits of the byte are received as zero when transmitting a one, the device has lost arbitration and must cease transmission of any additional bits of the current byte or transaction.

According to the preferred embodiment of the invention, a serial bus transaction consists of the transmission of a series of packets. The transaction begins with a transmission by the transaction initiator, which specifies the target network, device, length, type and payload of the transaction request. The transaction terminates with a packet having a type field in a specified range. As a result, all devices connected to the serial bus should monitor the serial data signal to determine when transactions begin and end. A serial bus network may have multiple simultaneous transactions occurring, however, so long as the target and initiator network addresses are all disjoint.

Parallel Processing

In one preferred embodiment of the invention, two or more general purpose media processors 12 can be linked together to achieve a multiple processor system. According to this embodiment, general purpose media processors 12 are linked together using their high bandwidth interface channels 124, either directly or through external switching components (not shown). The dual-master pair configuration described above can thus be extended for use in multiple-master ring configurations. Preferably, internal daemons provide for the generation of memory references to remote processors, accesses to local physical memory space, and the transport of remote references to other remote processors. In a multi-processor environment, all general purpose media processors 12 run off of a common clock frequency, as required by the communication channels 156 that connect between processors.

Referring to FIG. 18, each general purpose media processor 12 preferably includes at least a pair of inter-processor links 218 (see also FIG. 16(b)). In one configuration, both pairs of inter-processor links 218 can be connected between the two processors 12 to further enhance bandwidth. As shown in FIG. 18(a) several processors 12 may be interconnected in a linear network employing the

26

transponder daemons in each processor. In an alternate embodiment shown in FIG. 18(b), the inter-processor links 218 may be used to join the general purpose media processors 12 in a ring configuration. Alternatively still, general purpose media processors 12 may be interconnected into a two-dimensional network of processors of arbitrary size, as shown in FIG. 18(c). Sixteen processors are connected in FIG. 18(c) by connecting four ring networks. In yet another alternate embodiment, by connecting the inter-processor links 218 to external switching devices (not shown), multiprocessors with a large number of processors can be constructed with an arbitrary interconnection topology.

The requester, responder and transponder daemons preferably handle all inter-processor operations. When one general purpose media processor 12 attempts a load or store to a physical address of a remote processor, the requester daemon autonomously attempts to satisfy the remote memory reference by communicating with the external device. The external device may comprise another processor 12 or a switching device (not shown) that eventually reaches another processor 12. Preferably, two requester daemons are provided each processor 12, which act concurrently on two different byte channels and/or module addresses. The responder daemon accepts writes from a specified channel and module address, which enables an external device to generate transaction requests in local memory or to generate processor events. The responder daemon also generates link level writes to the same external device that communicated responses for the received transaction request. Two such responder daemons are preferably provided; each of which operate concurrently to two different byte channels and/or module addresses.

The transponder daemon accepts writes from a specified channel and module address, which enable an external device to cause a requester daemon to generate a request on another channel and module address. Preferably, two such transponder daemons are provided, each of which act concurrently (back-to-back) between two different byte channel and/or module addresses. As those skilled in the art will appreciate, the requester, responder and transponder daemons must act cooperatively to avoid deadlock that may arise due to an imbalance of requests in the system. Deadlocks prevent responses from being routed to their destinations, which may defeat the benefits of a multiprocessor distributed system.

According to one presently preferred embodiment of the invention, the general purpose media processor 12 can be implemented as one or more integrated circuit chips. Referring to FIG. 19, the presently preferred embodiment of the general purpose media processor 12 consists of a four-chip set. In the four-chip set, a general purpose media processor 12 is manufactured as a stand alone integrated circuit. The stand alone integrated circuit includes a memory management unit 122, instruction and data cache/buffers 118, 120, and an execution unit 100. A plurality of signal input/output pads 260 are provided around the circumference of the integrated circuit to communicate signals to and from the general purpose media processor 12 in a manner generally known in the art.

The second and third chips of the four-chip set comprise in an external memory element 158 and a channel interface device 216. The external memory element 158 includes an interface to the communication channel 156, a cache 150 and a memory interface 152. The channel interface device 216 also includes an interface to the communication channel 156, as well as buffer memory 262, and input/output interfaces 264. Both the external memory element 158 and the

5,809,321

27

channel interface device 216 include a plurality of input/output signal pads 260 to communicate signals to and from these devices in a generally known manner.

The fourth integrated circuit chip comprises a switch 226, which allows for installation of the general purpose media processor 12 in the heterogeneous network 38. In addition to the plurality of input/output pads 260, the switch 226 includes an interface to the communication channel 156. The switch 226 also preferably includes a buffer 262, a router 266, and a switch interface 268.

As those skilled in the art will appreciate, many implementations for the general purpose media processor 12 are possible in addition to the four-chip implementation described above. Rather than an integrated approach, the general purpose media processor can be implemented in a discrete manner. Alternatively, the general purpose media processor 12 can be implemented in a single integrated circuit, or in an implementation with fewer than four integrated circuit chips. Other combinations and permutations of these implementations are contemplated.

There has been described a system for processing streams of media data at substantially peak rates to allow for real time communication over a large heterogeneous network. The system includes a media processor at its core that is capable of processing such media data streams. The heterogeneous network consists of, for example, the fiber optic/coaxial cable/twisted wire network in place throughout the U.S. To provide for such communication of media data, a media processor according to the invention is disposed at various locations throughout the heterogeneous network. The media processor would thus function both in a server capacity and at an end user site within the network. Examples of such end user sites include televisions, set-top converter boxes, facsimile machines, wireless and cellular telephones, as well as large and small business and industrial applications.

To achieve such high rates of data throughput, the media processor includes an execution unit, high bandwidth interface, memory management unit, and pipelined instruction and data paths. The high bandwidth interface includes a mechanism for transmitting media data streams to and from the media processor at rates at or above the gigahertz frequency range. The media data stream can consist of transmission, presentation and storage type data transmitted alone or in a unified manner. Examples of such data types include audio, video, radio, network and digital communications. According to the invention, the media processor is dynamically partitionable to process any combination or permutation of these data types in any size.

A programmable, general purpose media processor system presents significant advantages over current multimedia communications. Rather than rigid, costly and inefficient specialized processors, the media processor provides a general purpose instruction set to ease programmability in a single device that is capable of performing all of the operations of the specialized processor combination. Providing a uniform instruction set for all media related operations eliminates the need for a programmer to learn several different instruction sets, each for a different specialized processor. The complexity of programming the specialized processors to work together and communicate with one another is also greatly reduced. The unified instruction set is also more efficient. Highly specialized general calculation instructions that are tailored to general or special types of calculations rather than enhancing communication are eliminated.

28

Moreover, the media processor system can be easily reprogrammed simply by transmitting or downloading new software over the network. In the specialized processor approach, new programming usually requires the delivery and installation of new hardware. Reprogramming the media processor can be done electronically, which of course is quicker and less costly than the replacement of hardware.

It is to be understood that a wide range of changes and modifications to the embodiments described above will be apparent to those skilled in the art and are contemplated. It is therefore intended that the foregoing detailed description be regarded as illustrative rather than limiting, and that it be understood that it is the following claims, including all equivalents, that are intended to define the spirit and scope of this invention.

We claim:

1. A system for unified media processing comprising:

a plurality of general purpose media processors, each media processor being operable at sustained peak data rates and having a dynamically partitioned execution unit, wherein a plurality of media data streams are concurrently transmitted over a single data path and are dynamically partitioned according to an elemental symbol width that is equal to or narrower than the data path, and having a high bandwidth interface, the high bandwidth interface coupled to external memory and input/output elements to receive and transmit data to the media processor at substantially peak rates; and

a bi-directional communication fabric, the plurality of media processors coupled to the bi-directional communication fabric to transmit and receive at least one media stream comprising presentation, transmission, and storage media information; and

wherein each media processor further comprises dedicated memory and wherein each of the plurality of media processors can employ any unused portion of the dedicated memory of another media processor in a shared manner to efficiently store and retrieve presentation, transmission and storage media information at substantially peak data rates.

2. The system defined in claim 1, wherein the bi-directional communication fabric comprises a fiber optic network.

3. The system defined in claim 1, wherein the bi-directional communication fabric comprises a heterogeneous network.

4. The system defined in claim 1, wherein the bi-directional communication fabric comprises a coaxial cable network.

5. The system defined in claim 1, wherein the bi-directional communication fabric comprises a wireless network.

6. The system defined in claim 1, wherein a subset of the plurality of media processors comprise network servers.

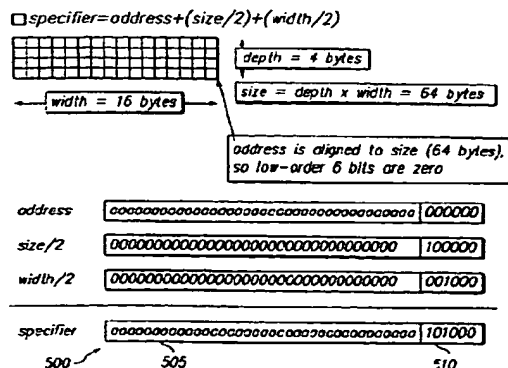
7. The system defined in claim 1, wherein the plurality of media processors are programmable by downloading program information over the bi-directional communication fabric.

8. The system defined in claim 1, wherein the plurality of media processors can access an idle execution unit of another media processor in a shared manner to efficiently process presentation, transmission and storage media information at substantially peak data rates.

* * * * *

EXHIBIT 9

(10) Patent No.: US 6,725,356 B2
(45) Date of Patent: *Apr. 20, 2004



US 6,725,356 B2

Page 2

U.S. PATENT DOCUMENTS

4,727,505 A	2/1988	Konishi et al.	
4,876,660 A	10/1989	Owens et al.	
4,893,267 A	1/1990	Alsup et al.	
4,956,801 A	9/1990	Priem et al.	
4,969,118 A	11/1990	Montoye et al.	
4,975,868 A	12/1990	Freerksen	
5,032,865 A	7/1991	Schlunt	
5,157,388 A	10/1992	Kohn	
5,201,056 A	4/1993	Daniel et al.	
5,268,855 A	12/1993	Mason et al.	
5,268,995 A	12/1993	Diefendorff et al.	
5,280,598 A *	1/1994	Osaki et al.	710/127
5,408,581 A	4/1995	Suzuki et al.	
5,423,051 A	6/1995	Fuller et al.	
5,426,600 A	6/1995	Nakagawa et al.	
5,487,024 A *	1/1996	Girardeau, Jr.	708/606
5,500,811 A	3/1996	Corry	
5,557,724 A	9/1996	Sampat et al.	
5,588,152 A	12/1996	Dapp et al.	
5,592,405 A	1/1997	Gove et al.	
5,600,814 A *	2/1997	Gahan et al.	711/100
5,640,543 A	6/1997	Farrell et al.	
5,642,306 A	6/1997	Mennemeier et al.	
5,666,298 A	9/1997	Peleg et al.	
5,669,010 A	9/1997	Duluk, Jr.	
5,673,407 A	9/1997	Poland et al.	
5,675,526 A	10/1997	Peleg et al.	
5,680,338 A	10/1997	Agarwal et al.	
5,721,892 A	2/1998	Peleg et al.	
5,734,874 A	3/1998	Van Hook et al.	
5,757,432 A	5/1998	Dulong et al.	
5,758,176 A	5/1998	Agarwal et al.	
5,768,546 A *	6/1998	Kwon	710/127
5,802,336 A	9/1998	Peleg et al.	
5,809,292 A	9/1998	Wilkinson et al.	
5,818,739 A	10/1998	Peleg et al.	
5,825,677 A	10/1998	Agarwal et al.	
5,835,782 A	11/1998	Liu et al.	
5,886,732 A	3/1999	Humpleman	
5,922,066 A	7/1999	Cho et al.	
5,983,257 A	11/1999	Dulong et al.	
6,016,538 A	1/2000	Guttag et al.	
6,092,094 A	7/2000	Ireton	
6,295,599 B1 *	9/2001	Hansen et al.	712/32
6,401,194 B1	6/2002	Nguyen et al.	

OTHER PUBLICATIONS

IBM Creates PowerPC Processors for AS/400, Two New CPU's Implement 64-Bit Power PC with Extensions by Linley Gwennap, Jul. 31, 1995.

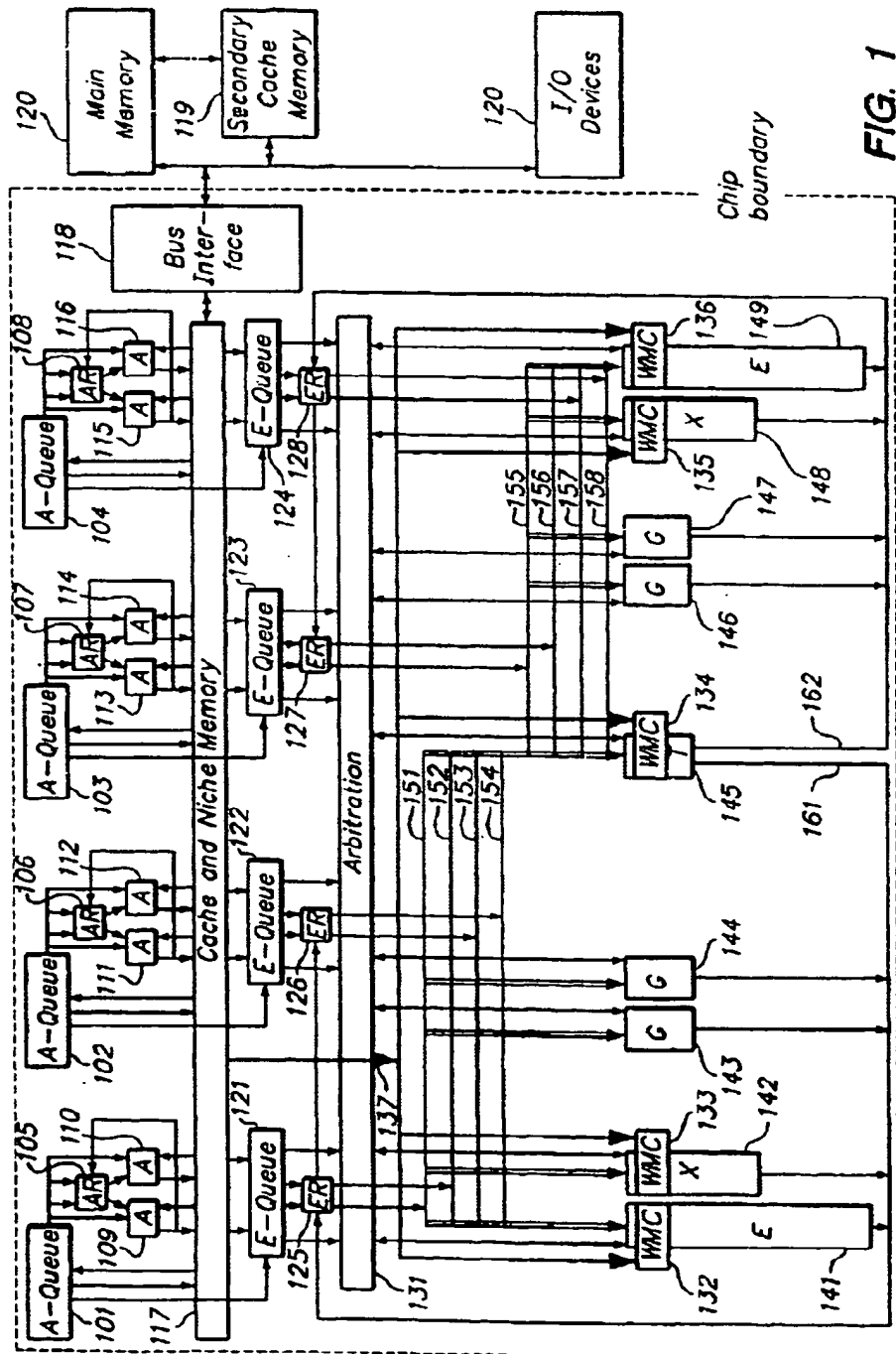
The Visual Instruction Set (VSI) in UltraSPAR™, L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, G. Zyner, May 3, 1995.

Osborne McGraw-Hill, i860™ Microprocessor Architecture, Neal Margulis, Foreword by Les Kohn.

A General-Purpose Array Processor for Seismic Processing, Nov.-Dec., 1984, vol. 1, No. 3) Revisiting past digital signal processor technology, Don Shaver- Jan.-Mar. 1998.

Accelerating Multimedia with Enhanced Microprocessors, Ruby B. Lee, 1995.

* cited by examiner



U.S. Patent

Apr. 20, 2004

Sheet 2 of 148

US 6,725,356 B2

$$\square rd_{128} = m[rc](128*64/size) * rb_{128}$$

$$m[rc](128*64/size)$$

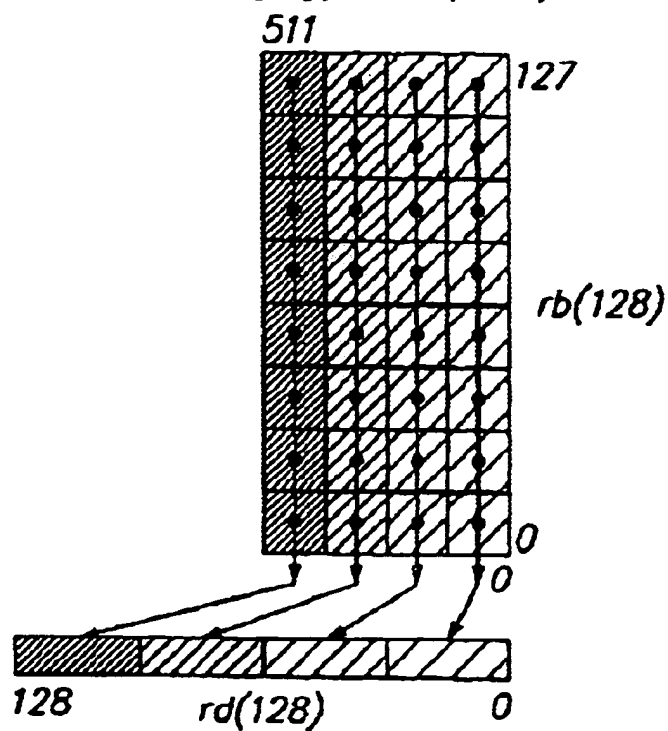


FIG. 2

U.S. Patent

Apr. 20, 2004

Sheet 3 of 148

US 6,725,356 B2

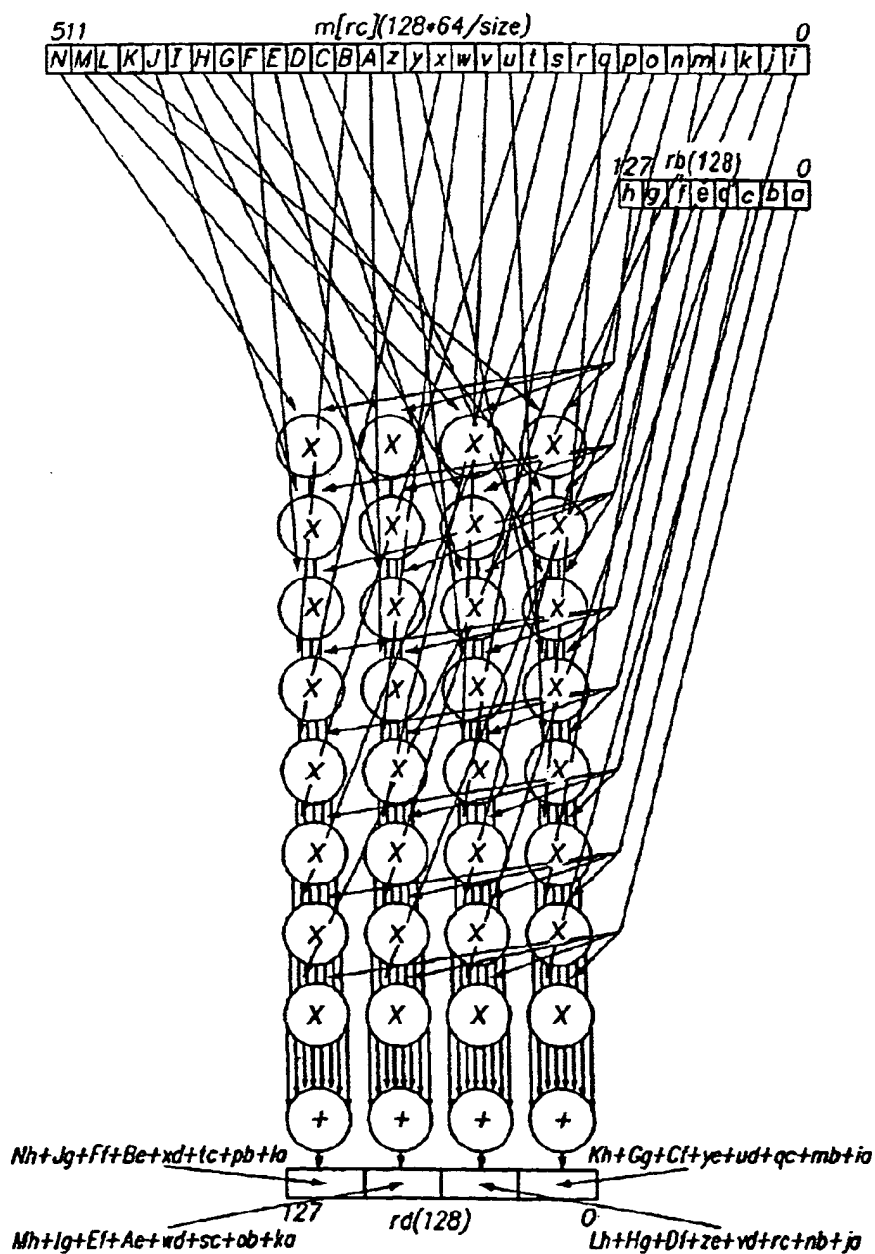


FIG. 3

U.S. Patent

Apr. 20, 2004

Sheet 4 of 148

US 6,725,356 B2

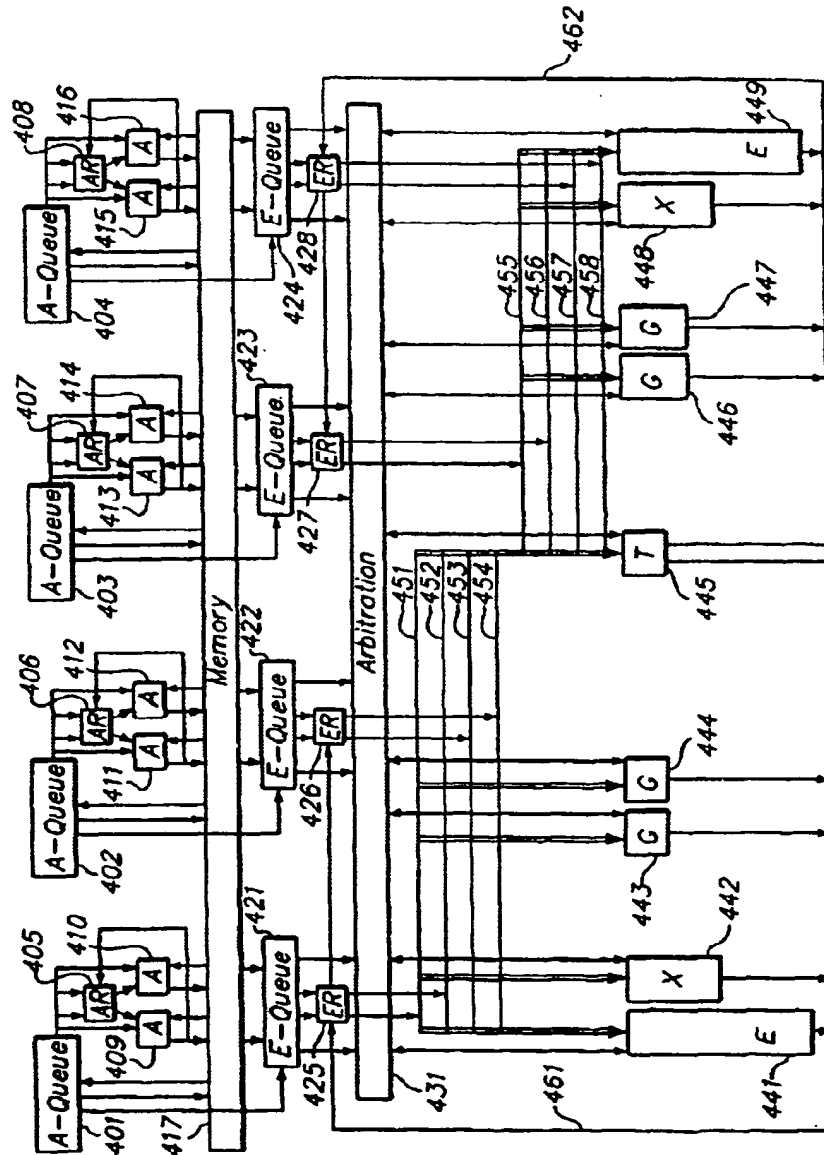


FIG. 4

□ $specifier = address + (size/2) + (width/2)$

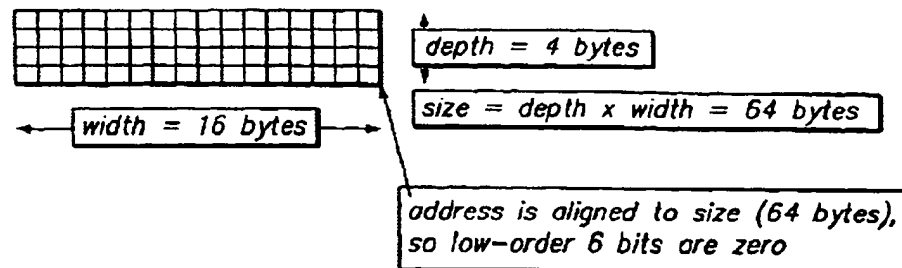
[illegible]

FIG. 5

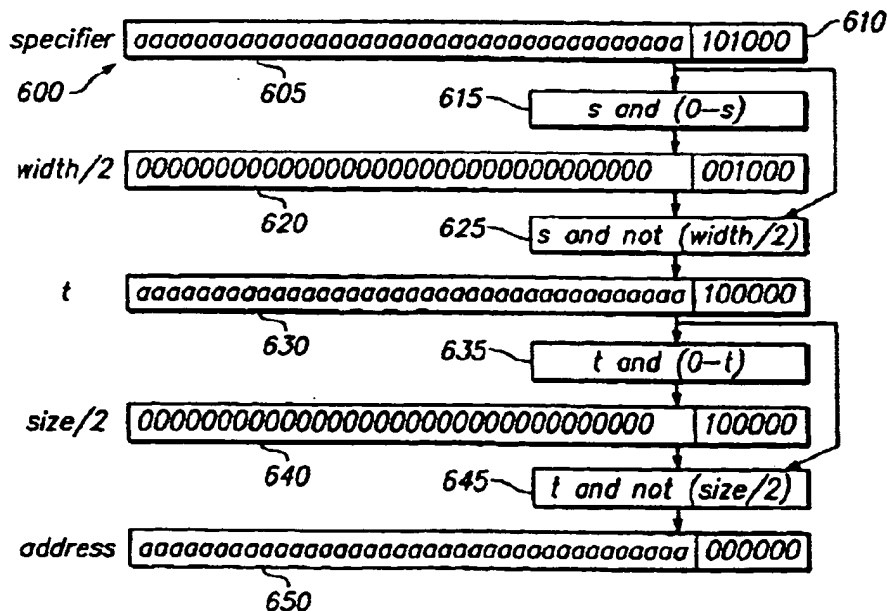


FIG. 6

U.S. Patent

Apr. 20, 2004

Sheet 6 of 148

US 6,725,356 B2

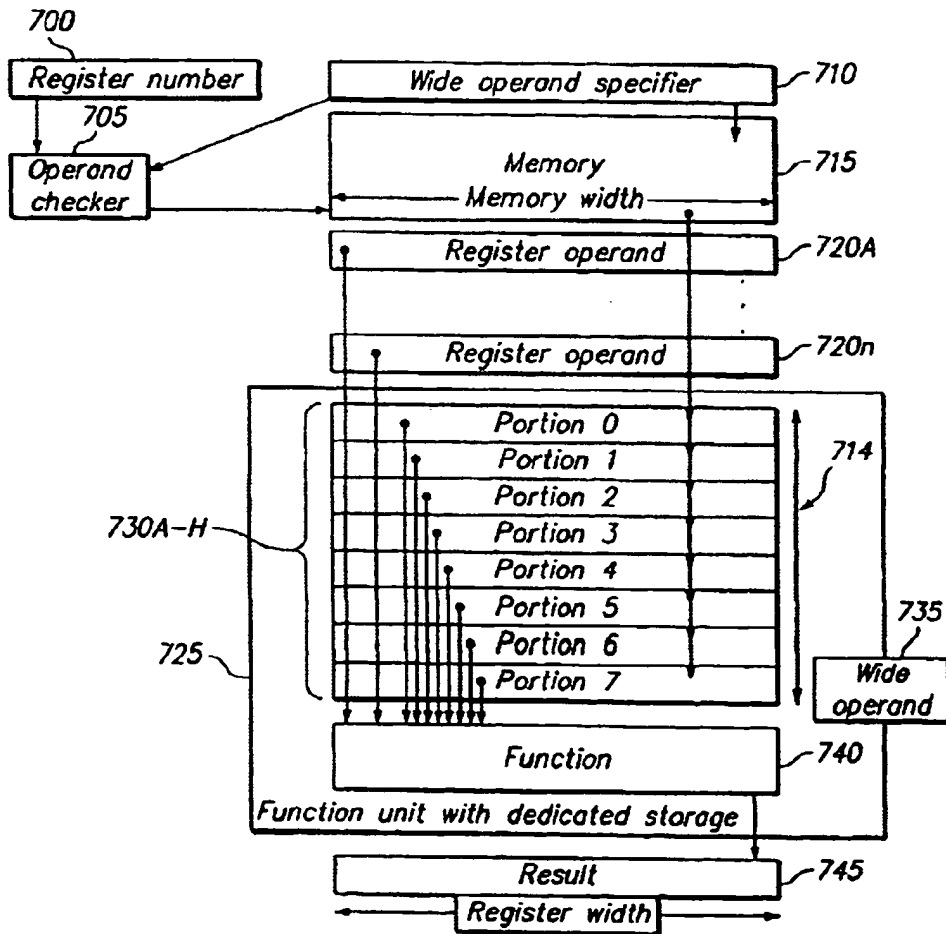


FIG. 7

U.S. Patent

Apr. 20, 2004

Sheet 7 of 148

US 6,725,356 B2

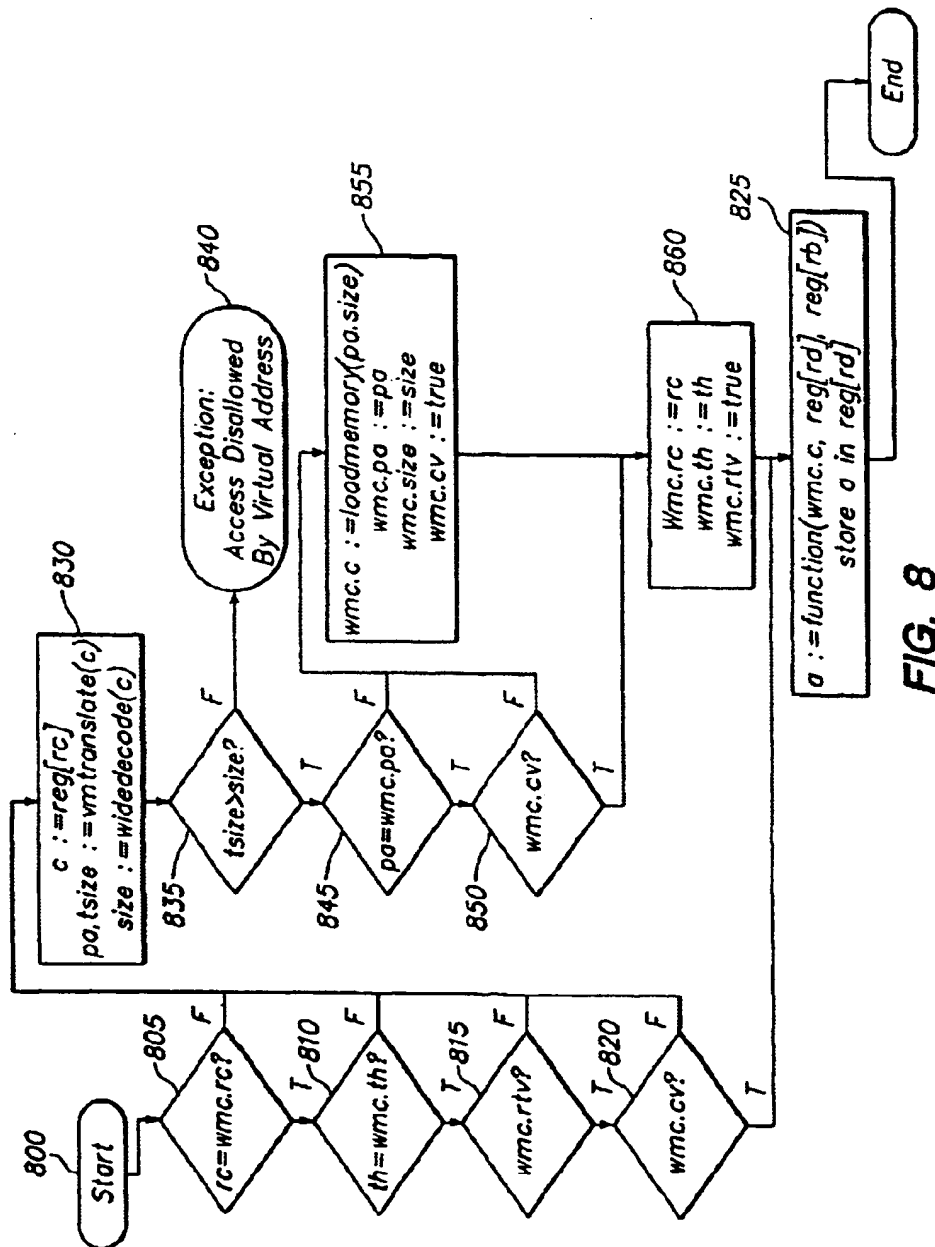


FIG. 8

U.S. Patent

Apr. 20, 2004

Sheet 8 of 148

US 6,725,356 B2

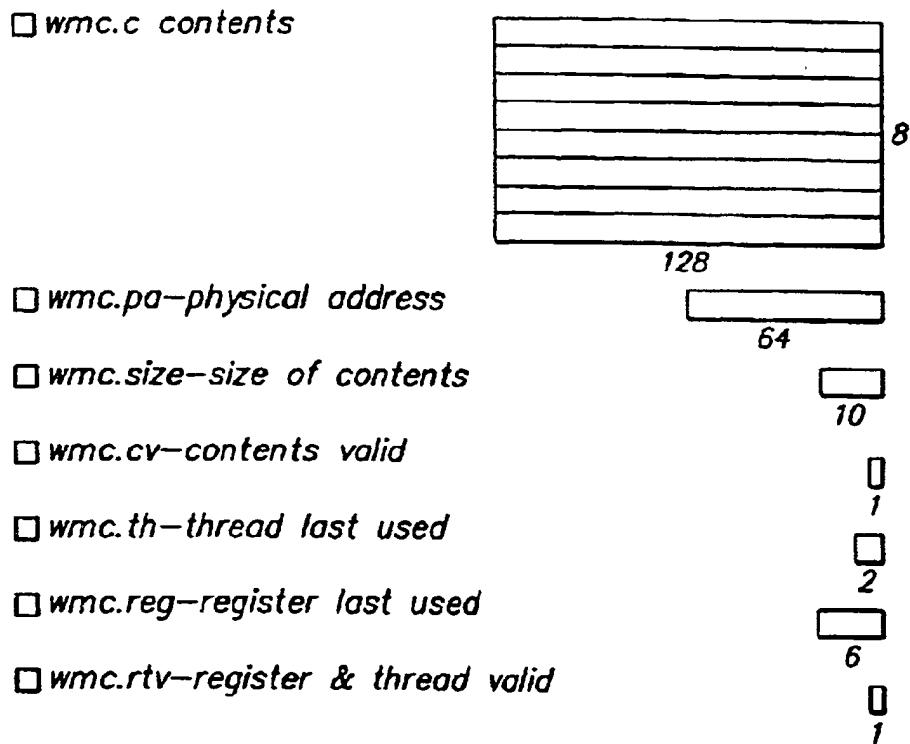


FIG. 9

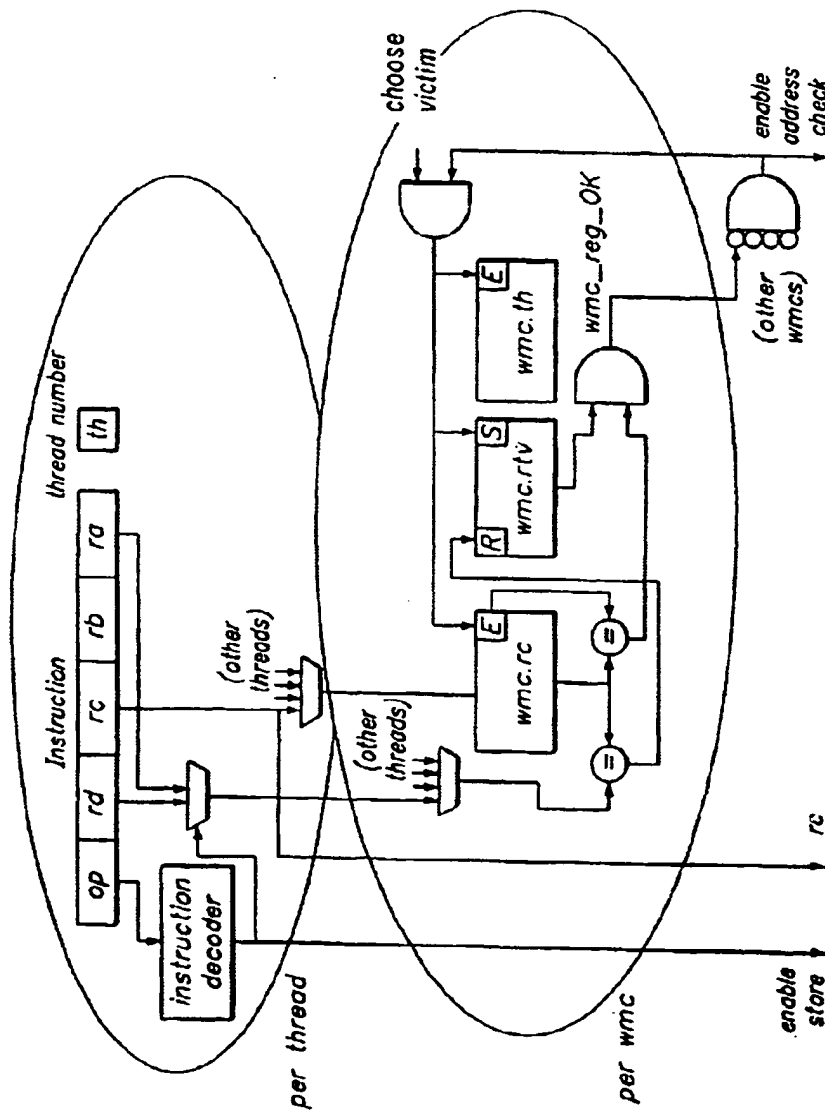
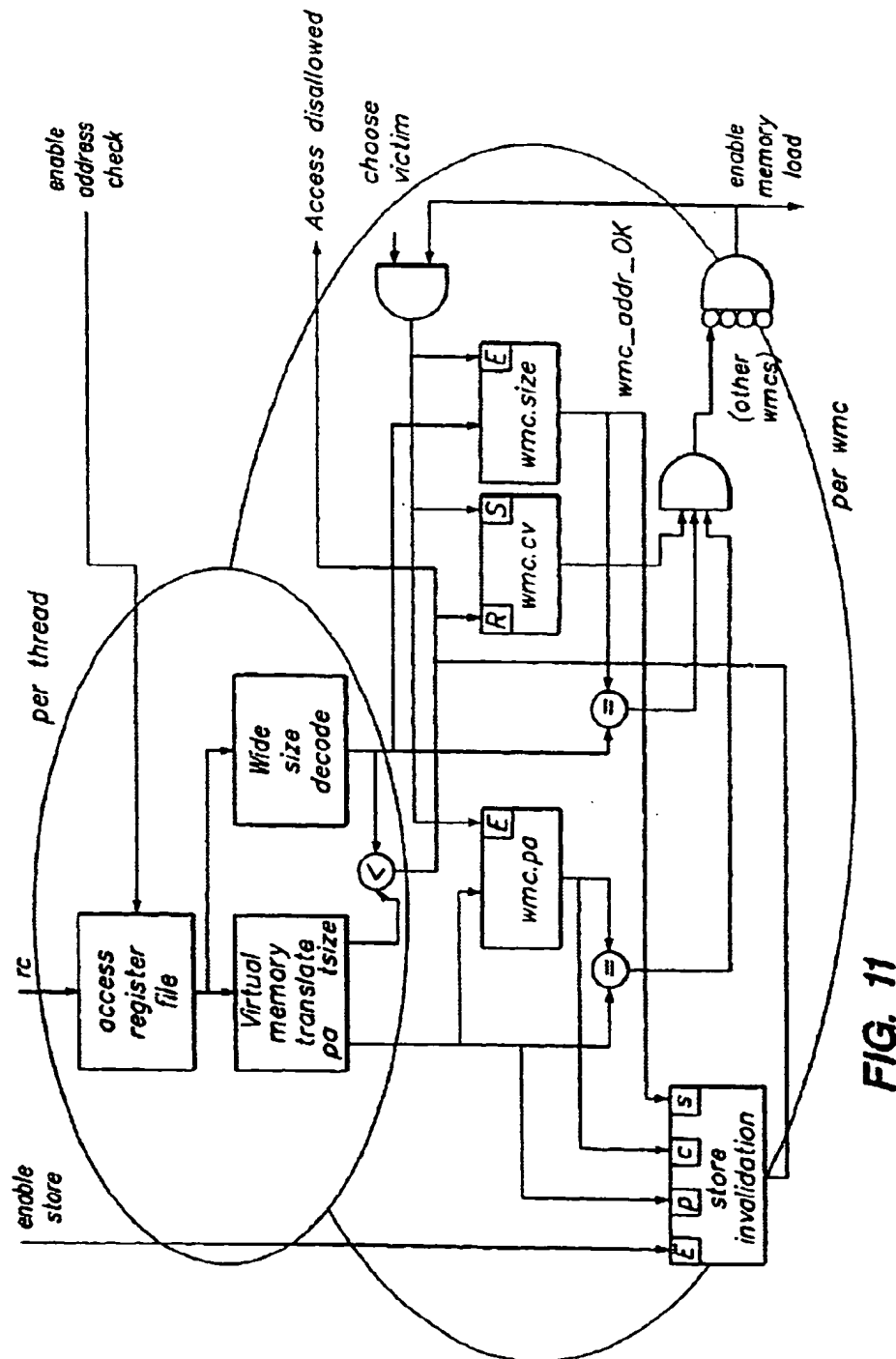


FIG. 10



U.S. Patent

Apr. 20, 2004

Sheet 11 of 148

US 6,725,356 B2210
↙Operation codes

W.SWITCH.B	Wide switch big-endian
W.SWITCH.L	Wide switch little-endian

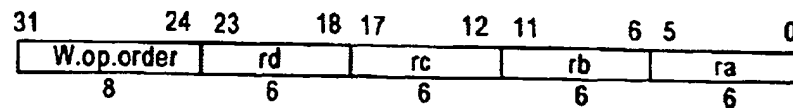
Selection

class	op	order
Wide switch	W.SWITCH	B L

Format

W.op.order ra=rc,rd,rb

ra=woporder(rc,rd,rb)

**FIG. 12A**

U.S. Patent

Apr. 20, 2004

Sheet 12 of 148

US 6,725,356 B2

1230

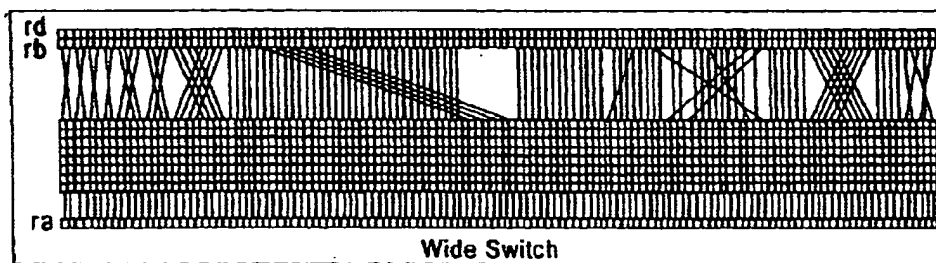


FIG. 12B

U.S. Patent

Apr. 20, 2004

Sheet 13 of 148

US 6,725,356 B2

1250

Definition

```

defWideSwitch(op,rd,rc,rb,ra)
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 64)
  b ← RegRead(rb, 128)
  if c1..0 ≠ 0 then
    raise AccessDisallowedByVirtual Address
  elseif c6..0 ≠ 0 then
    VirtAddr ← c and (c-1)
    W ← wsize ← (c and (0-c)) || 01
  else
    VirAddr ← c
    w ← wsize ← 128
  endif
  msize ← 8*wsize
  lwsiz ← log(wsize)
  case op of
    W.SWITCH.B:
      order ← B
    W.SWITCH.L:
      order ← L
  endcase
  m ← LoadMemory(c, VirtAddr,msize,order)
  db ← d || b
  for i ← 0 to 127
    j ← 0 || i1wsiz-1..0
    k ← m7*w+j || m6*w+j || m5*w+j || m4*w+j || m3*w+j || m2*w+j || mw+j || mj
    l ← i7..1wsiz || j1wsiz-1..0
    ai ← dbl
  endfor
  RegWrite(ra, 128, a)
enddef

```


FIG. 12C

U.S. Patent

Apr. 20, 2004

Sheet 14 of 148

US 6,725,356 B2

1280


Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 12D

U.S. Patent

Apr. 20, 2004

Sheet 15 of 148

US 6,725,356 B2

1210

Operation codes

W.TRANSLATE.8.B	Wide translate bytes big-endian
W.TRANSLATE.16.B	Wide translate doublets bit-endian
W.TRANSLATE.32.B	Wide translate quadlets bit-endian
W.TRANSLATE.64.B	Wide translate octlets big-endian
W.TRANSLATE.8.L	Wide translate bytes little-endian
W.TRANSLATE.16.L	Wide translate doublets little-endian
W.TRANSLATE.32.L	Wide translate quadlets little-endian
W.TRANSLATE.64.L	Wide translate octlets little-endian

Selection

class	size	order
Wide translate	8 16 32 64	B L

Format

W.TRANSLATE.size.order rd=rc,rb

rd=wtranslatesizeorder(rc,rb)

31	2434	1817	1211	65	21	0
W.TRANSLATE.order	rd	rc	rb	0	sz	
6	6	6	6	4	2	

sz ← log(size) = 3

FIG. 13A

U.S. Patent

Apr. 20, 2004

Sheet 16 of 148

US 6,725,356 B2

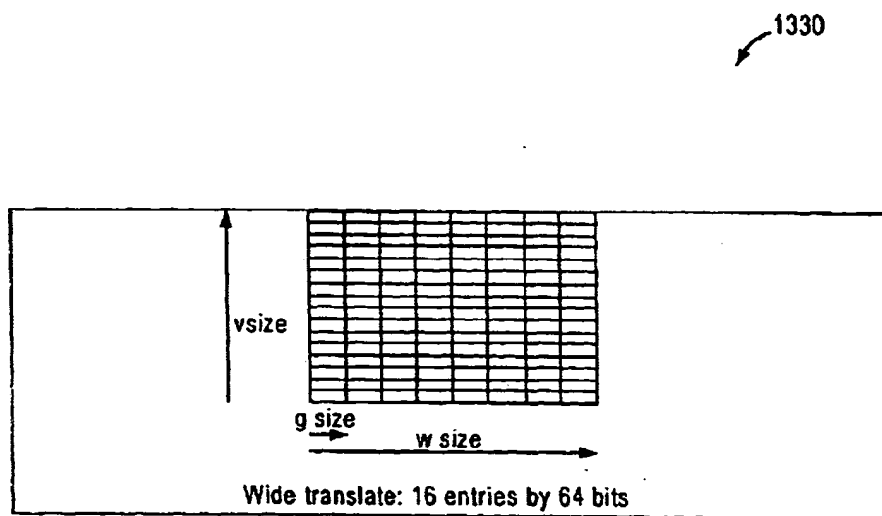


FIG. 13B

U.S. Patent

Apr. 20, 2004

Sheet 17 of 148

US 6,725,356 B2

1350

Definition

```

def Wide Translate(op, gsize, rd, rc, rb)
  c ← RegRead(rc, 64)
  b ← RegRead(rb, 128)
  lsize ← log(gsize)
  if clsize-4..0 ≠ 0 then
    raise AccessDisallowedByVirtual Address
  endif
  if c4..lsize-3 ≠ 0 then
    wsize ← (c and (0-c)) || 03
    t ← c and (c-1)
  else
    wsize ← 128
    t ← c
  endif
  lsize ← log(wsize)
  if tlsize+4..lsize-2 ≠ 0 then
    msize ← (t and (0-t)) || 04
    VirtAddr ← t and (t-1)
  else
    msize ← 256*wsize
    VirtAddr ← t
  endif
  case op of
    W.TRANSLATE.B:
      order ← B
    W.TRANSLATE.L:
      order ← L
  endcase
  m ← LoadMemory(c, VirtAddr, msize, order)
  vsize ← msize/wsize
  lsize ← log(vsize)
  for i ← 0 to 128-gsize by gsize
    j ← ((order=B)lsize )(bvsize-1+i..i ) * wsize + ilsize-1..0
    agsize-1+i..i ← mj+gsize-1..j
  endfor
  RegWrite(rd, 128, a)
enddef

```

FIG. 13C

U.S. Patent

Apr. 20, 2004

Sheet 18 of 148

US 6,725,356 B2

1380



Exceptions

Access disallowed by virtual address

Access disallowed by tag

Access disallowed by global TB

Access disallowed by local TB

Access detail required by tag

Access detail required by local TB

Access detail required by global TB

Local TB miss

Global TB miss

FIG. 13D

U.S. Patent

Apr. 20, 2004

Sheet 19 of 148

US 6,725,356 B2

Operation codes

1410

W.MUL.MAT.8.B	Wide multiply matrix signed byte big-endian
W.MUL.MAT.8.L	Wide multiply matrix signed byte little-endian
W.MUL.MAT.16.B	Wide multiply matrix signed doublet big-endian
W.MUL.MAT.16.L	Wide multiply matrix signed doublet little-endian
W.MUL.MAT.32.B	Wide multiply matrix signed quadlet big-endian
W.MUL.MAT.32.L	Wide multiply matrix signed quadlet little-endian
W.MUL.MAT.C.8.B	Wide multiply matrix signed complex byte big-endian
W.MUL.MAT.C.8.L	Wide multiply matrix signed complex byte little-endian
W.MUL.MAT.C.16.B	Wide multiply matrix signed complex doublet big-endian
W.MUL.MAT.C.16.L	Wide multiply matrix signed complex doublet little-endian
W.MUL.MAT.M.8.B	Wide multiply matrix mixed-signed byte big-endian
W.MUL.MAT.M.8.L	Wide multiply matrix mixed-signed byte little-endian
W.MUL.MAT.M.16.B	Wide multiply matrix mixed-signed doublet big-endian
W.MUL.MAT.M.16.L	Wide multiply matrix mixed-signed doublet little-endian
W.MUL.MAT.M.32.B	Wide multiply matrix mixed-signed quadlet big-endian
W.MUL.MAT.M.32.L	Wide multiply matrix mixed-signed quadlet little-endian
W.MUL.MAT.P.8.B	Wide multiply matrix polynomial byte big-endian
W.MUL.MAT.P.8.L	Wide multiply matrix polynomial byte little-endian
W.MUL.MAT.P.16.B	Wide multiply matrix polynomial doublet big-endian
W.MUL.MAT.P.16.L	Wide multiply matrix polynomial doublet little-endian
W.MUL.MAT.P.32.B	Wide multiply matrix polynomial quadlet big-endian
W.MUL.MAT.P.32.L	Wide multiply matrix polynomial quadlet little-endian
W.MUL.MAT.U.8.B	Wide multiply matrix unsigned byte big-endian
W.MUL.MAT.U.8.L	Wide multiply matrix unsigned byte little-endian
W.MUL.MAT.U.16.B	Wide multiply matrix unsigned doublet big-endian
W.MUL.MAT.U.16.L	Wide multiply matrix unsigned doublet little-endian
W.MUL.MAT.U.32.B	Wide multiply matrix unsigned quadlet big-endian
W.MUL.MAT.U.32.L	Wide multiply matrix unsigned quadlet little-endian

Selection

class	op	type	size	order
multiply	W.MUL.MAT	NONE MUP	8 16 32	B
				L
		C	8 16	B
				L

Format

W.op.size.order rd=rc,rb

rd=wopsizeorder(rc,rb)

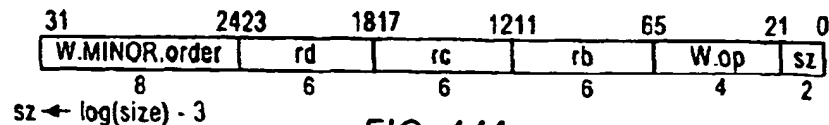


FIG. 14A

U.S. Patent

Apr. 20, 2004

Sheet 20 of 148

US 6,725,356 B2

1430

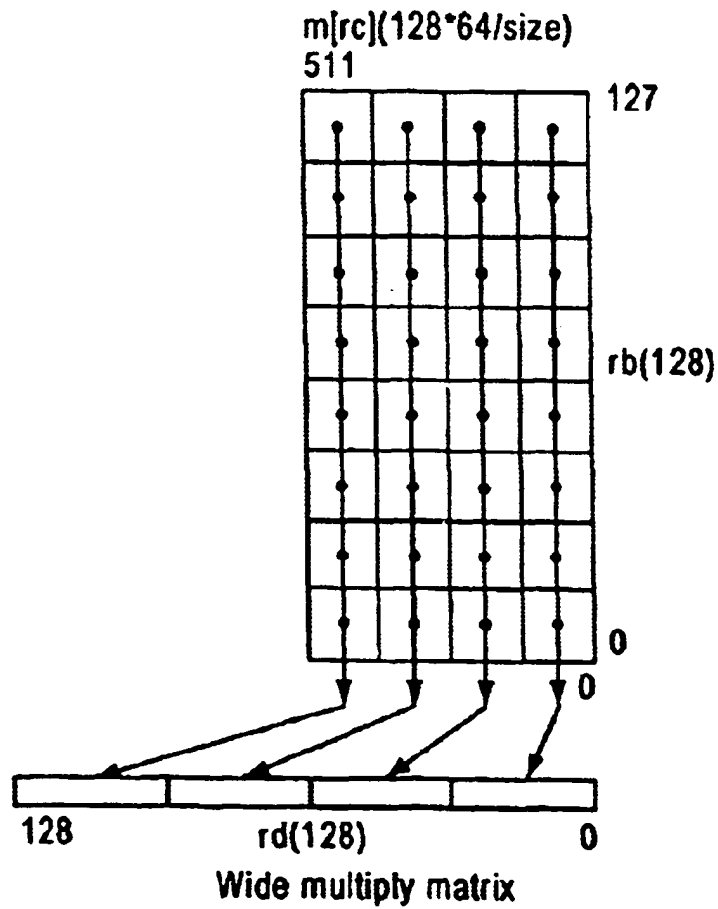


FIG. 14B

U.S. Patent

Apr. 20, 2004

Sheet 21 of 148

US 6,725,356 B2

1460

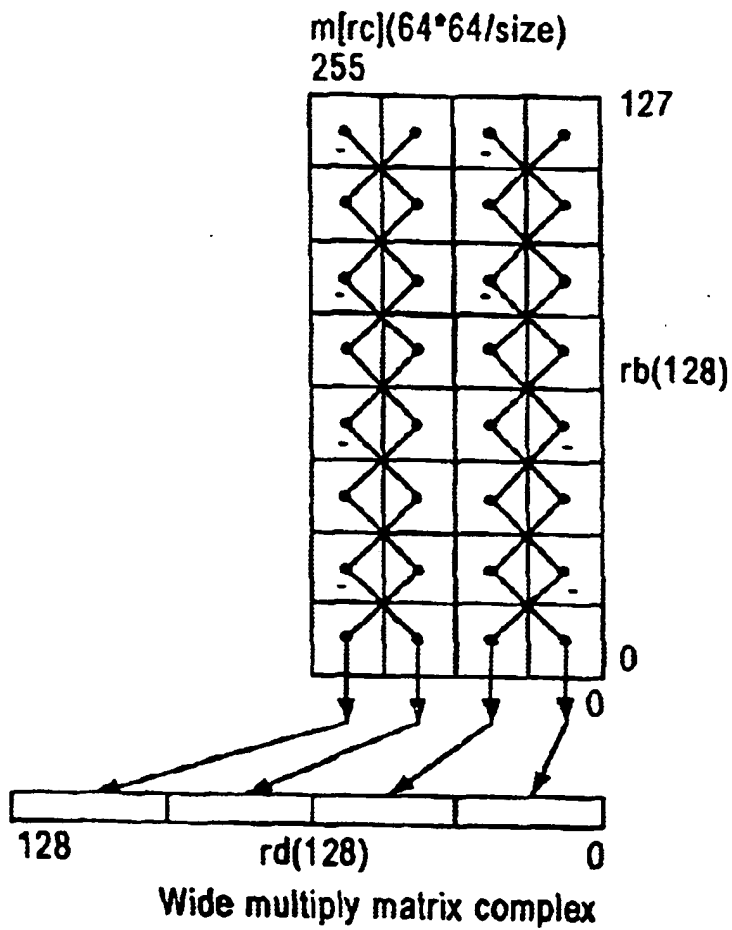


FIG. 14C

U.S. Patent

Apr. 20, 2004

Sheet 22 of 148

US 6,725,356 B2

Definition

1480

```

def mul(size,h,vs,v,i,ws,i) as
  mul ← ((vs&vsize-1+i)h-size || vsize-1+i..i) * ((ws&ws-1+j)h-size || ws-1+j..j)
enddef

```

```

def c ← PolyMultiply(size,a,b) as
  p[0] ← 02*size
  for k ← 0 to size-1
    p[k+1] ← p[k] ^ ak ? (0size-k || b) : 02*size
  endfor
  c ← p[size]
enddef

```

```

def WideMultiplyMatrix(major,op,gsize,rd,rc,rb)
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 64)
  b ← RegRead(rb,128)
  lgsize ← log(gsize)
  if clgsize-4..0 ≠ 0 then
    raise AccessDisallowedByVirtualAddress
  endif
  if c2..lgsize-3 ≠ 0 then
    wsize ← (c and (0-c)) || 04
    t ← c and (c-1)
  else
    wsize ← 64
    t ← a
  endif
  lsize ← log(wsize)
  if tlsize+6-lgsize..lsize-3 ≠ 0 then
    msize ← (t and (0-t)) || 04
    VirtAddr ← t and (t-1)
  else
    msize ← 128*wsize/gsize
    VirtAddr ← t
  endif
  case major of
    W.MINOR.B:
      order ← B
    W.MINOR.L:
      order ← L
  endcase
enddef

```

FIG. 14D-1

U.S. Patent

Apr. 20, 2004

Sheet 23 of 148

US 6,725,356 B2

1480

```

case op of
  M.MUL.MAT.U.8, W.MUL.MAT.U.16, W.MUL.MAT.U.32,
  W.MUL.MAT.U.64:
    ms ← bs ← 0
  W.MUL.MAT.M.8, W.MUL.MAT.M.16, W.MUL.MAT.M.32,
  W.MUL.MAT.M.64:
    ms ← 0
    bs ← 1
  W.MUL.MAT.8, W.MUL.MAT.16, W.MUL.MAT.32,
  W.MUL.MAT.64, W.MUL.MAT.C.8, W.MUL.MAT.C.16,
  W.MUL.MAT.C.32, W.MUL.MAT.C.64:
    ms ← bs ← 1
  W.MUL.MAT.P.8, W.MUL.MAT.P.16, W.MUL.MAT.P.32,
  W.MUL.MAT.P.64:
endcase
m ← LoadMemory(c,VirtAddr,msize,order)
h ← 2*gsiz

for i ← 0 to wsize-gsize by gsize
  q[0] ← 02*gsiz
  for j ← 0 to vsize-gsize by gsize
    case op of
      W.MUL.MAT.P.8, W.MUL.MAT.P.16,
      W.MUL.MAT.P.32, W.MUL.MAT.P.64:
        k ← i+wsize*j8..lgsiz
        q[j+gsiz] ← q[j] ^ PolyMultiply(gsiz,mk+gsiz-1..k,
        bj+gsiz-1..j)
      W.MUL.MAT.C.8, W.MUL.MAT.C.16, W.MUL.MAT.C.32,
      W.MUL.MAT.C.64:
        if (~i) & gsiz = 0 then
          k ← i-(j&gsiz)+wsize*j8..lgsiz+1
          q[j+gsiz] ← q[i] + mul(gsiz,h,ms,m,k,bs,b,j)
        else
          k ← i+gsiz+wsize*j8..lgsiz+1
          q[i+gsiz] ← q[i] = mul(gsiz,h,ms,m,k,bs,b,j)
        endif
  endfor
endfor

```

FIG. 14D-2

```

W.MUL.MAT.8, W.MUL.MAT.16, W.MUL.MAT.32,
W.MUL.MAT.64, W.MUL.MAT.M.8, W.MUL.MAT.M.16,
W.MUL.MAT.M.32, W.MUL.MAT.M.64, W.MUL.MAT.U.8,
W.MUL.MAT.U.16, W.MUL.MAT.U.32, W.MUL.MAT.U.64
    q[i+gsize] ← q[i] + mul(gsize,h,ms,m,i+wsz*
        j8..lgsize,bs,b,i)
endfor
    a2*gsz-1+2*i..2*i ← q[vsize]
endfor
a127..2*wsz ← 0
RegWrite(rd, 128, a)
enddef

```

FIG. 14D-3

U.S. Patent

Apr. 20, 2004

Sheet 25 of 148

US 6,725,356 B2

1490



Exceptions

Access disallowed by virtual address

Access disallowed by tag

Access disallowed by global TB

Access disallowed by local TB

Access detail required by tag

Access detail required by local TB

Access detail required by global TB

Local TB miss


Global TB miss

FIG. 14E

U.S. Patent

Apr. 20, 2004

Sheet 26 of 148

US 6,725,356 B2
 1510
Operation codes

W.MUL.MAT.X.B	Wide multiply matrix extract big-endian
W.MUL.MAT.X.L	Wide multiply matrix extract little-indian

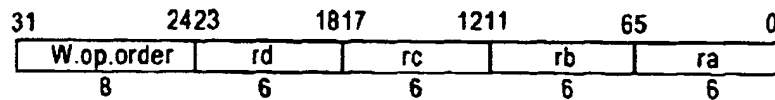
Selection

class	op	order
Multiply matrix extract	W.MUL.MAT.X	B L

Format

W.op.order ra=rc,rd,rb

ra=wop(rc,rd,rb)

**FIG. 15A**

U.S. Patent

Apr. 20, 2004

Sheet 27 of 148

US 6,725,356 B2

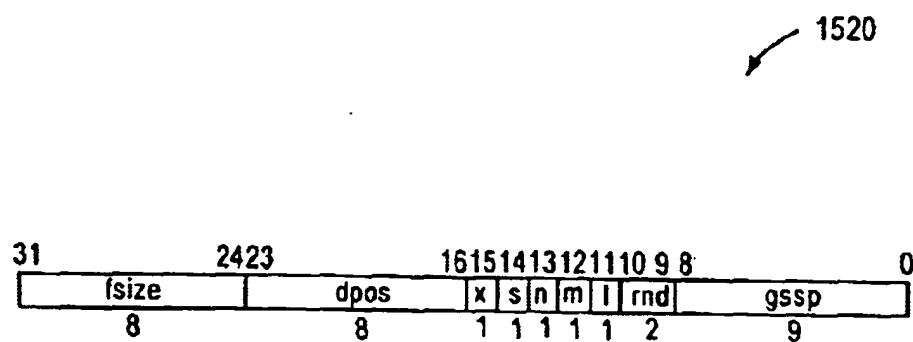


FIG. 15B

U.S. Patent

Apr. 20, 2004

Sheet 28 of 148

US 6,725,356 B2

1530

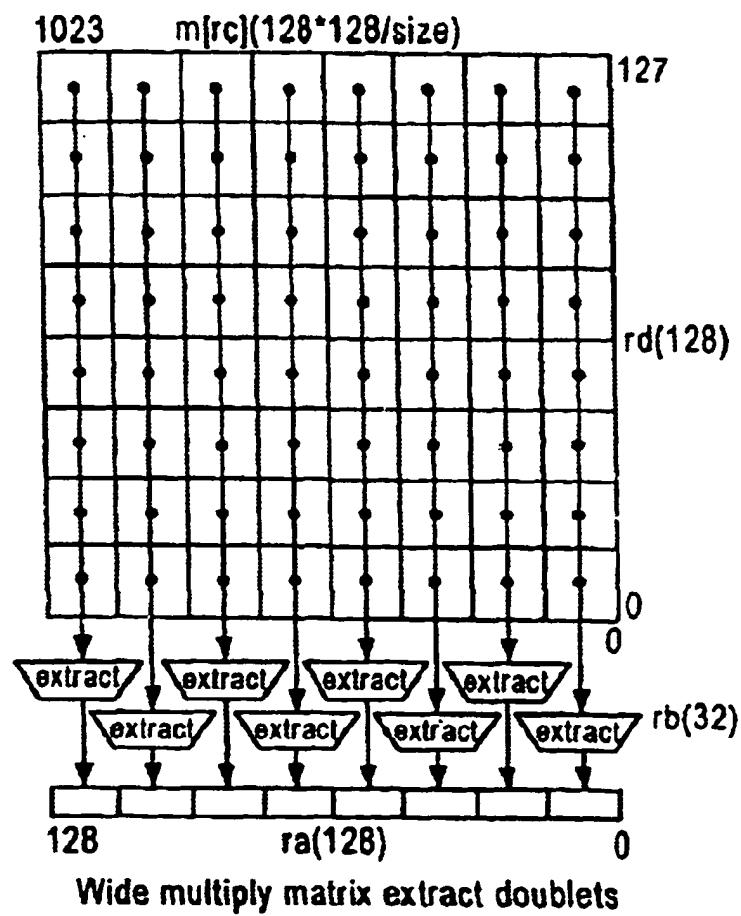


FIG. 15C

U.S. Patent

Apr. 20, 2004

Sheet 29 of 148

US 6,725,356 B2

1560

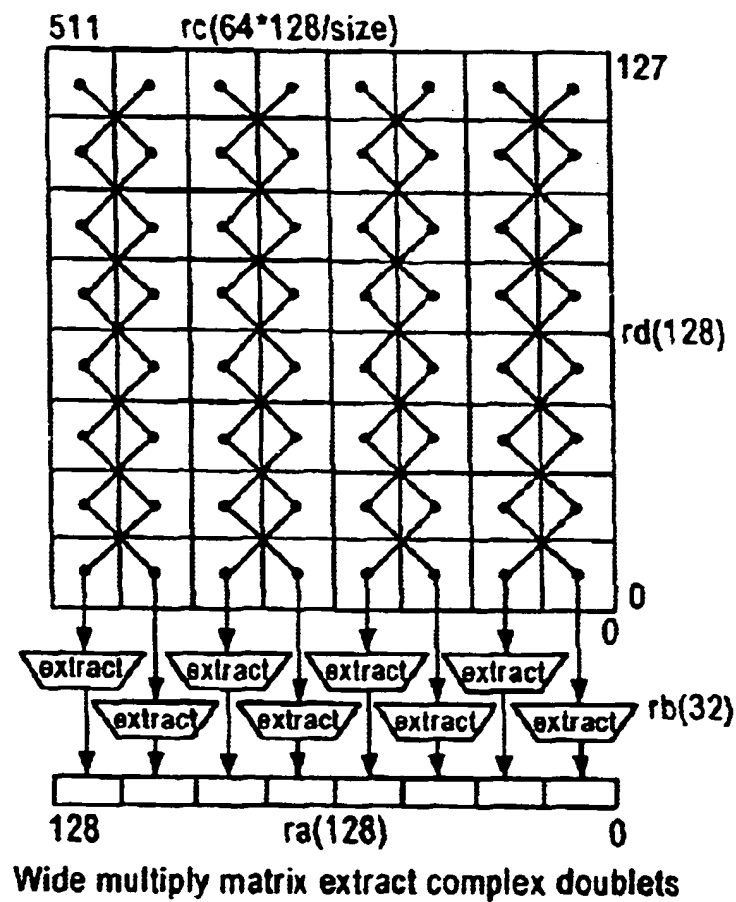


FIG. 15D

U.S. Patent

Apr. 20, 2004

Sheet 30 of 148

US 6,725,356 B2

Definition

1580

```

def mul(size,h,vs,v,i,ws,w,j) as
    mul ← ((vs&vsize-1+i)h-size || vsize-1+i..i) * ((ws&wsize-1+j)h-size || wsize-1+j..j)
enddef

def WideMultiplyMatrixExtract(op,ra,rb,rc,rd)
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    case b8_0 of
        0..255:
            ssize ← 128
        256..383:
            ssize ← 64
        384..447:
            ssize ← 32
        448..479:
            ssize ← 16
        480..495:
            ssize ← 8
        496..503:
            ssize ← 4
        504..507:
            ssize ← 2
        508..511:
            ssize ← 1
    endcase
    l ← b11
    m ← b12
    n ← b13
    signed ← b14
    if c3_0 ≠ 0 then
        wsize ← (c and (0-c)) || 04
        t ← c and (c-1)
    else
        wsize ← 128
        t ← c
    endif
    if ssize < 8 then
        gsize ← 8
    elseif ssize > wsize/2 then
        gsize ← wsize/2
    else

```

FIG. 15E-1

U.S. Patent

Apr. 20, 2004

Sheet 31 of 148

US 6,725,356 B2

1580

```

    gsize ← sgsz
endif
lgsize ← log(gsize)
lwsize ← log(wsize)
if  $lwsize + 6 - n - lgsize - lwsize - 3 \neq 0$  then
    msize ←  $(t \text{ and } (0-t)) \parallel 0^4$ 
    VirtAddr ←  $t \text{ and } (t-1)$ 
else
    msize ←  $64 * (2-n) * wsize / gsize$ 
    VirtAddr ← t
endif
vsize ←  $(1+n) * msize * gsize / wsize$ 
mm ← LoadMemory(c, VirtAddr, msize, order)
lmsize ← log(msize)
if  $(VirtAddr_{lmsize-4..0} \neq 0)$  then
    raise AccessDisallowedByVirtualAddress
endif
case op of
    W.MUL.MAT.X.B:
        order ← B
    W.MUL.MAT.X.L:
        order ← L
endcase
ms ← signed
ds ← signed ^ m
as ← signed or m
spos ←  $(b_{8..0}) \text{ and } (2 * gsize - 1)$ 
dpos ←  $(0 \parallel b_{23..16}) \text{ and } (gsize - 1)$ 
r ← spos
sfsz ←  $(0 \parallel b_{31..24}) \text{ and } (gsize - 1)$ 
tfsz ←  $(sfsz = 0) \text{ or } ((sfsz + dpos) > gsize) ? gsize - dpos : sfsz$ 
fsz ←  $(tfsz + spos > h) ? h - spos : tfsz$ 
if  $(b_{10..9} = Z) \ \& \ \sim \text{signed}$  then
    rnd ← F
else
    rnd ←  $b_{10..9}$ 
endif

```

FIG. 15E-2

U.S. Patent

Apr. 20, 2004

Sheet 32 of 148

US 6,725,356 B2

1580

```

for i ← 0 to wsize-gsize by gsize
  q[0] ← 02*gsizex7-lgsizex
  for j ← 0 to vsize-gsize by gsize
    if n then
      if (~) & j & gsize = 0 then
        k ← i-(j&gsizex)+wsize*j8..lgsizex+1
        q[i+gsizex] ← q[i] + mul(gsizex,h,ms,mm,k,ds,d,j)
      else
        k ← i+gsizex+wsize*j8..lgsizex+1
        q[i+gsizex] ← q[i] - mul(gsizex,h,ms,mm,k,ds,d,j)
      endif
    else
      q[i+gsizex] ← q[i] = mul(gsizex,h,ms,mm,i+j*wsize/gsizex,ds,d,j)
    endif
  endfor
  p ← q[128]
  case rnd of
    none, N:
      s ← 0h-r || -pr || pr-1
    Z:
      s ← 0h-r || pr-1
    F:
      s ← 0h
    C:
      s ← 0h-r || 1r
  endcase
  v ← ((ds & ph-1) || p) + (0 || s)

  if (vh..r+fsizex = (as & vr+fsizex-1)h+1-r-fsizex) or not 1 then
    w ← (as & vr+fsizex-1)gsizex-fsizex-dpos || vfsizex-1+r..r || 0dpos
  else
    w ← (s ? (vh || ~vhgsizex-dpos-1) : 1gsizex-dpos) || 0dpos
  endif
  asize-1+i..i ← w
endfor
a127..wsize ← 0
RegWrite(ra, 128, a)
enddef

```

FIG. 15E-3

U.S. Patent

Apr. 20, 2004

Sheet 33 of 148

US 6,725,356 B2

1570



Exceptions

Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 15F

U.S. Patent

Apr. 20, 2004

Sheet 34 of 148

US 6,725,356 B2

1610

Operation codes

W.MUL.MAT.X.I.8.B	Wide multiply matrix extract immediate signed byte big-endian
W.MUL.MAT.X.I.8.L	Wide multiply matrix extract immediate signed byte little-endian
W.MUL.MAT.X.I.16.B	Wide multiply matrix extract immediate signed doublet big-endian
W.MUL.MAT.X.I.16.L	Wide multiply matrix extract immediate signed doublet little-endian
W.MUL.MAT.X.I.32.B	Wide multiply matrix extract immediate signed quadlet big-endian
W.MUL.MAT.X.I.32.L	Wide multiply matrix extract immediate signed quadlet little-endian
W.MUL.MAT.X.I.64.B	Wide multiply matrix extract immediate signed octlets big-endian
W.MUL.MAT.X.I.64.L	Wide multiply matrix extract immediate signed octlets little-endian
W.MUL.MAT.X.I.C.8.B	Wide multiply matrix extract immediate complex bytes big-endian
W.MUL.MAT.X.I.C.8.L	Wide multiply matrix extract immediate complex bytes little-endian
W.MUL.MAT.X.I.C.16.B	Wide multiply matrix extract immediate complex doublets big-endian
W.MUL.MAT.X.I.C.16.L	Wide multiply matrix extract immediate complex doublets little-endian
W.MUL.MAT.X.I.C.32.B	Wide multiply matrix extract immediate complex quadlets big-endian
W.MUL.MAT.X.I.C.32.L	Wide multiply matrix extract immediate complex quadlets little-endian

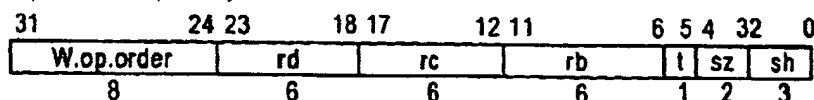
Selection

class	op	type	size	order
wide multiply extract immediate	W.MUL.MAT.X.I	NONE	8 16 32 64	L B
		C	8 16 32	L B

Format

W.op.tsizerd=rc,rb,i

rd=wopsizeorder(rc,rb,i)



sz ← log(size) - 3

assert size+3 ≥ i ≥ size-4

sh ← i - size

FIG. 16A

U.S. Patent

Apr. 20, 2004

Sheet 35 of 148

US 6,725,356 B2

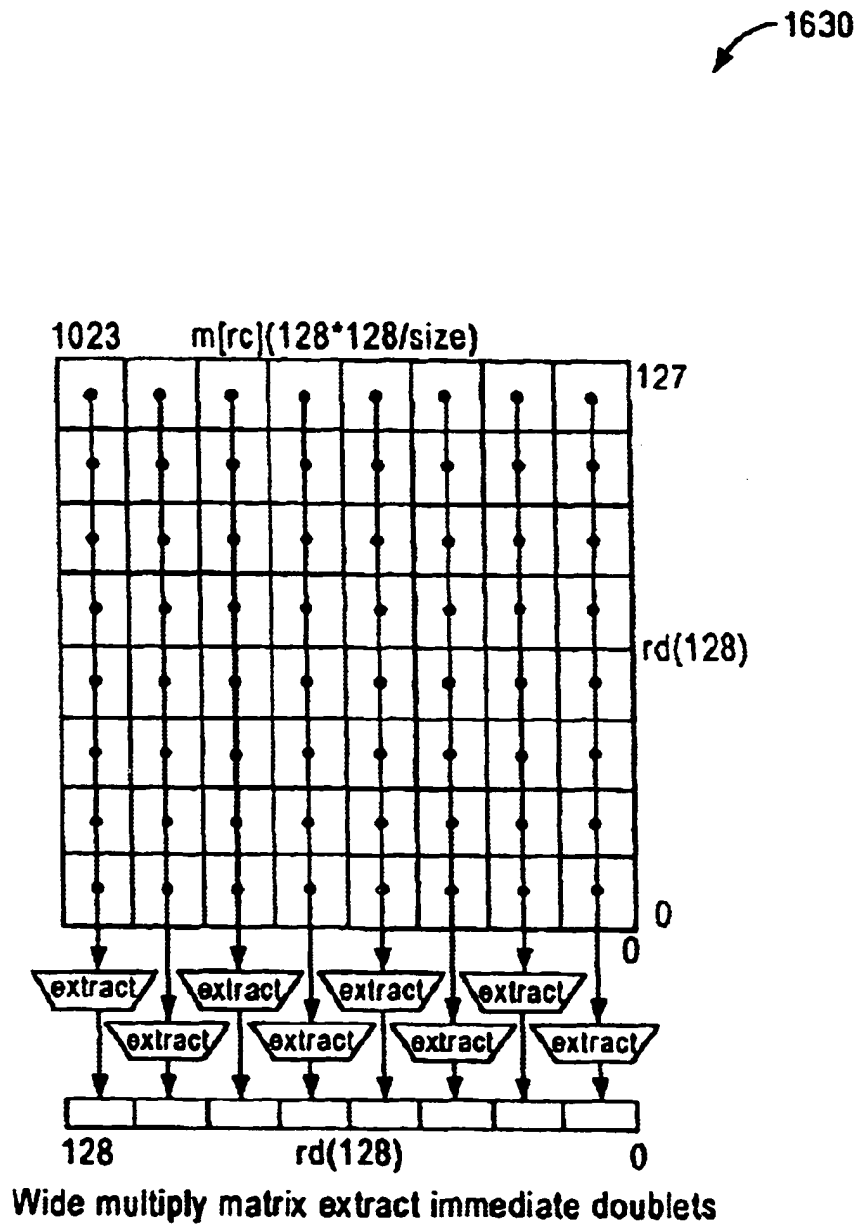


FIG. 16B

U.S. Patent

Apr. 20, 2004

Sheet 36 of 148

US 6,725,356 B2

1660

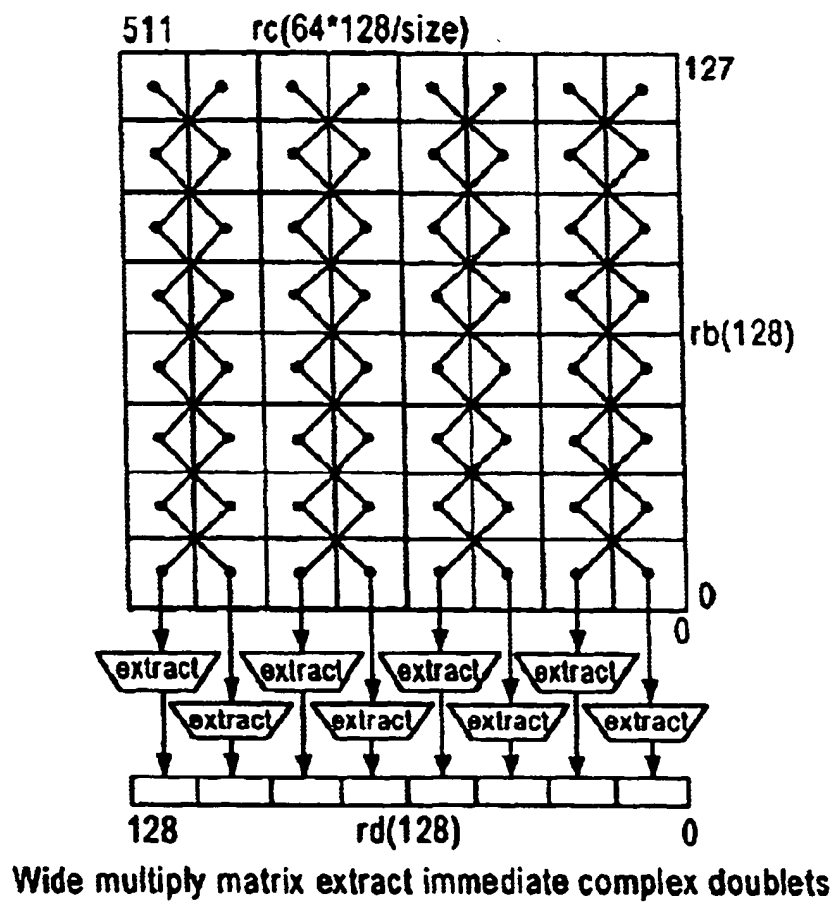


FIG. 16C

U.S. Patent

Apr. 20, 2004

Sheet 37 of 148

US 6,725,356 B2

Definition

1680

```

def mul(size,h,vs,v,i,ws,w,j) as
  mul ← ((vs&vsize-1+i)h-size || vsize-1+i..i) * ((ws&wsiz-1+j)h-size || wsiz-1+j..j)
endef

```

```

def WideMultiplyMatrixExtractimmediate(op,type,gsize,rd,rc,rb,sh)

```

```

  c ← RegRead(rc, 64)
  b ← RegRead(rb, 128)
  lsize ← log(gsize)
  case type of
    NONE:
      if cgsize-4..0 ≠ 0 then
        raise AccessDisallowedBy VirtualAddress
      endif
      if c3..lsize-3 ≠ 0 then
        wsize ← (c and (0-c)) || 04
        t ← c and (c-1)
      else
        wsize ← 128
        t ← c
      endif
      lwsiz ← log(wsize)
      if tlwsiz+6-lsize..lsize-3 ≠ 0 then
        msiz ← (t and (0-1)) || 04
        VirtAddr ← t and (t-1)
      else
        msiz ← 128*wsize/gsize
        VirtAddr ← t

```

```

  C:
    if cgsize-4..0 ≠ 0 then
      raise AccessDisallowedByVirtualAddress
    endif
    if c3..lsize-3 ≠ 0 then
      wsize ← (c and (0-c)) || 04
      t ← c and (c-1)
    else
      wsize ← 128
      t ← c
    endif
    lwsiz ← log(wsize)
    if tlwsiz+5-lsize..lsize-3 ≠ 0 then
      msiz ← (t and (0-1)) || 04

```

FIG. 16D-1

U.S. Patent

Apr. 20, 2004

Sheet 38 of 148

US 6,725,356 B2

```

VirtAddr ← t and (t-1)
else
  msize ← 64*wsz/gsize
  VirtAddr ← t
endif
vsize ← 2*msz*gsz/wsz
endcase
case of of
  W.MUL.MAT.X.I.B:
    order ← B
  W.MUL.MAT.X.I.L:
    order ← L
endcase
as ← ms ← bs ← 1
m ← LoadMemory(c,VirtAddr,msize,order)
h ← (2*gsz) + 7 - lgsz-(ms and bs)
r ← gsz + (sh5||sh)
for ← 0 to wsz-gsz by gsz
  q[0] ← 02*gsz+7-lgsz
  for j ← 0 to vsize-gsz by gsz
    case type of
      NONE:
        q[j+gsz] ← q[j] + mul(gsz,h,ms,m,i+wsz*
          i8..lgsz,bs,b,j)
      C:
        if (-i) & j & gsz = 0 then
          k ← i-(j&gsz)+wsz*j8..lgsz+1
          q[j+gsz] ← q[j] + mul(gsz,h,ms,m,k,bs,b,j)
        else
          k ← i+gsz+wsz*j8..lgsz+1
          q[j+gsz] ← q[j] - mul(gsz,h,ms,m,k,bs,b,j)
        endif
      endcase
    endfor
    p ← q[vsize]
    s ← 0h-r||~pr|| pr-1
    v ← ((as & ph-1)||p) + (0||s)
    if (vh..r+gsz = (as & vr+gsz-1)h+1-r-gsz then
      agsz-1+i..i ← vgsz-1+r..r
    else
      agsz-1+i..i ← as ? (vh||~vhgsz-1) : lgsz
    endif
  endfor
  a127..wsz ← 0
  RegWrite(rd, 128, a)
enddef

```

1680


FIG. 16D-2

U.S. Patent

Apr. 20, 2004

Sheet 39 of 148

US 6,725,356 B2

1690


Exceptions

Access disallowed by virtual address

Access disallowed by tag

Access disallowed by global TB

Access disallowed by local TB

Access detail required by tag

Access detail required by local TB

Access detail required by global TB

Local TB miss

Global TB miss

FIG. 16E

U.S. Patent

Apr. 20, 2004

Sheet 40 of 148

US 6,725,356 B2

1710

Operation codes

W.MUL.MAT.C.F.16.B	Wide multiply matrix complex floating-point half big-endian
W.MUL.MAT.C.F.16.L	Wide multiply matrix complex floating-point little-endian
W.MUL.MAT.C.F.32.B	Wide multiply matrix complex floating-point single big-endian
W.MUL.MAT.C.F.32.L	Wide multiply matrix complex floating-point single little-endian
W.MUL.MAT.F.16.B	Wide multiply matrix floating-point half big-endian
W.MUL.MAT.F.16.L	Wide multiply matrix floating-point half little-endian
W.MUL.MAT.F.32.B	Wide multiply matrix floating-point single big-endian
W.MUL.MAT.F.32.L	Wide multiply matrix floating-point single little-endian
W.MUL.MAT.F.64.B	Wide multiply matrix floating-point double big-endian
W.MUL.MAT.F.64.L	Wide multiply matrix floating-point double little-endian

Selection

class	op	type	prec	order
wide multiply matrix	W.MUL.MAT	F	16 32 64	L B
		C.F	16 32	L B

Format

W.op.prec.order rd=rc,rb

rd=wopprecorder(rc,rb)

31	24 23	18 17	12 11	6 5	21	0
W.MINOR.order	rd	rc	rb	W.op	pr	
8	6	6	6	4	2	

Pr ← log(prec) - 3

FIG. 17A

U.S. Patent

Apr. 20, 2004

Sheet 41 of 148

US 6,725,356 B2

1730

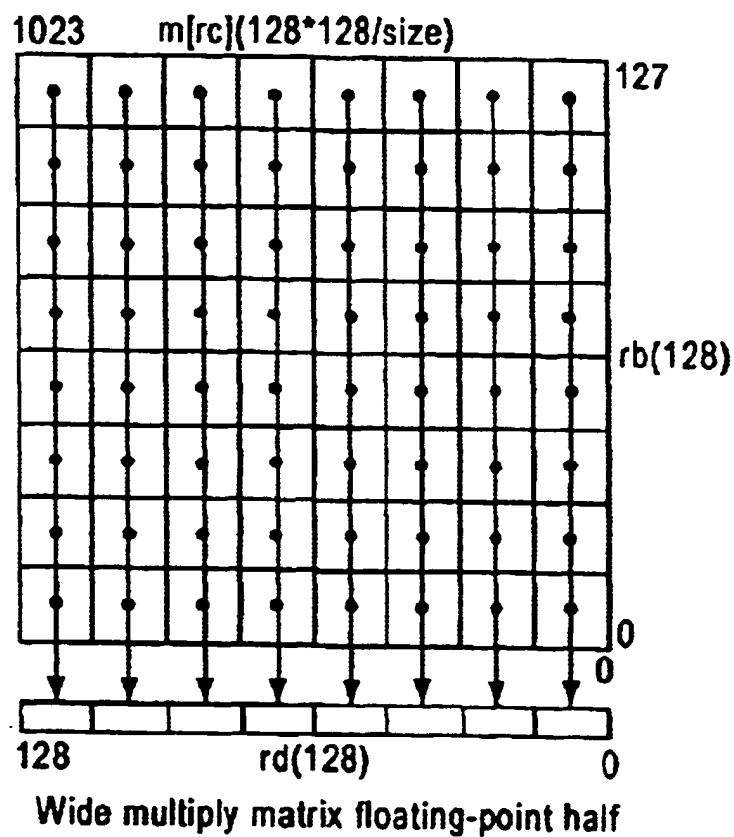


FIG. 17B

U.S. Patent

Apr. 20, 2004

Sheet 42 of 148

US 6,725,356 B2

1760

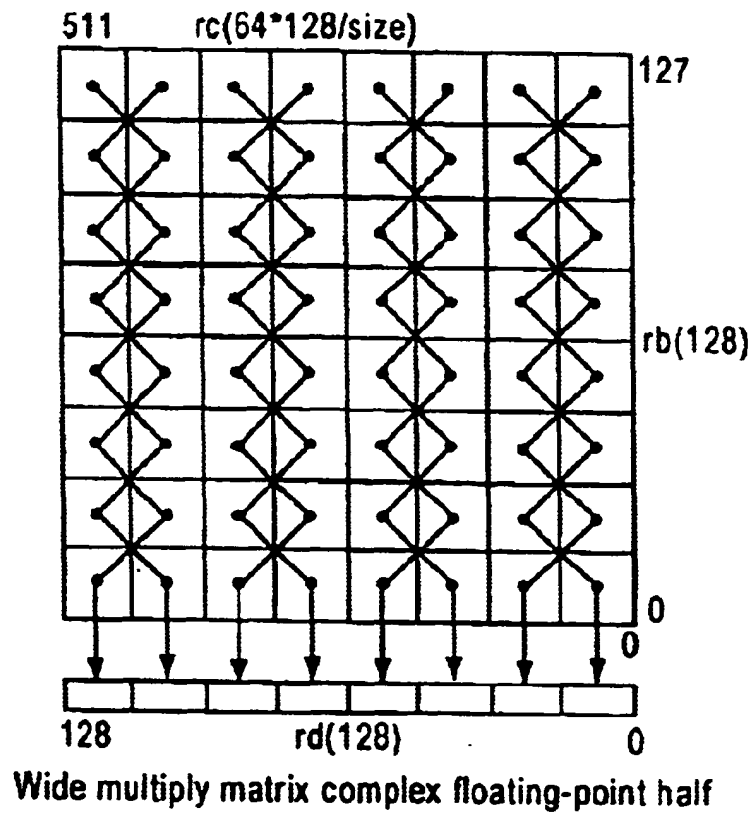


FIG. 17C

U.S. Patent

Apr. 20, 2004

Sheet 43 of 148

US 6,725,356 B2

Definition

1780

```

def mul(size,v,i,w,j) as
    mul ← fmul(F(size,vsize-1+i..i),F(size,wsiz-1+j..j))
enddef

def WideMultiplyMatrixFloatingPoint(major,op,gsize,rd,rc,rb)
    c ← RegRead(rc, 64)
    b ← RegRead(rb, 128)
    lsize ← log(gsize)
    switch op of
        W.MUL.MAT.F.16, W.MUL.MAT.F.32, W.MUL.MAT.F.64:
            if clgsize-4..0 ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
            if c3..lgsize-3 ≠ 0 then
                wsize ← (c and (0-c)) || 04
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
            endif
            lwsiz ← log(wsize)
            if tlwsiz+6-lsize..lwsiz-3 ≠ 0 then
                msize ← (t and (0-t)) || 04
                VirtAddr ← t and (t-1)
            else
                msize ← 128wsize/gsize
                VirtAddr ← t
            endif
            vsize ← msizegsize/wsize
        W.MUL.MAT.C.F.16, W.MUL.MAT.C.F.32, W.MUL.MAT.C.F.64:
            if clgsize-4..0 ≠ 0 then
                raise AccessDisallowedByVirtualAddress
            endif
            if c3..lgsize-3 ≠ 0 then
                wsize ← (c and (0-c)) || 04
                t ← c and (c-1)
            else
                wsize ← 128
                t ← c
            endif
            lwsiz ← log(wsize)
            if tlwsiz+5-lsize..lwsiz-3 ≠ 0 then

```

FIG. 17D-1

U.S. Patent

Apr. 20, 2004

Sheet 44 of 148

US 6,725,356 B2

1780

```

        msize ← (t and (0-t)) || 04
        VirtAddr ← t and (t-1)
    else
        msize ← 64*wsiz/gsiz
        VirtAddr ← t
    endif
    vsiz ← 2*msiz*gsiz/wsiz
endcase
case major of
    M.MINOR.B:
        order ← B
    M.MINOR.L:
        order ← L
endcase
m ← LoadMemory(c,VirtAddr,msiz,order)
for i ← 0 to wsiz-gsiz by gsiz
    q[0].t ← NULL
    for j ← 0 to vsiz-gsiz by gsiz
        case op of
            W.MUL.MAT.F.16, W.MUL.MAT.F.32, W.MUL.MAT.F.64:
                q[j+gsiz] ← faddq[j], mul(gsiz,m,i+wsiz*
                    j8..lgsiz+1.b,j))
            W.MUL.MAT.C.F.16, W.MUL.MAT.C.F.32,
            W.MUL.MAT.C.F.64:
                if (~i) & j & gsiz = 0 then
                    k ← i-(j&gsiz)+wsiz*j8..lgsiz+1
                    q[j+gsiz] ← faqq[j], mul(gsiz,m,k,b,j))
                else
                    k ← i+gsiz+wsiz*j8..lgsiz+1
                    q[j+gsiz] ← fsubq[j], mul(gsiz,m,k,b,j))
                endif
        endcase
    endfor
    agsiz-1+i..i ← q[vsiz]
endfor
a127..wsiz ← 0
RegWrite(rd, 128, a)
enddef

```


FIG. 17D-2

U.S. Patent

Apr. 20, 2004

Sheet 45 of 148

US 6,725,356 B2

1780


Exceptions

Floating-point arithmetic
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 17E

U.S. Patent

Apr. 20, 2004

Sheet 46 of 148

US 6,725,356 B2

1810

Operation codes

W.MUL.MAT.G.8.B	Wide multiply matrix Galois bytes big-endian
W.MUL.MAT.G.8.L	Wide multiply matrix Galois bytes little-endian

Selection

class	op	size	order
Multiply matrix Galois	W.MUL.MAT.G	8	B L

Format

W.op.order ra=rc,rd,rb

ra=woporder(rc,rd,rb)

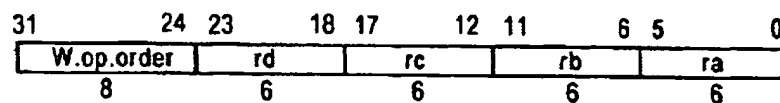


FIG. 18A

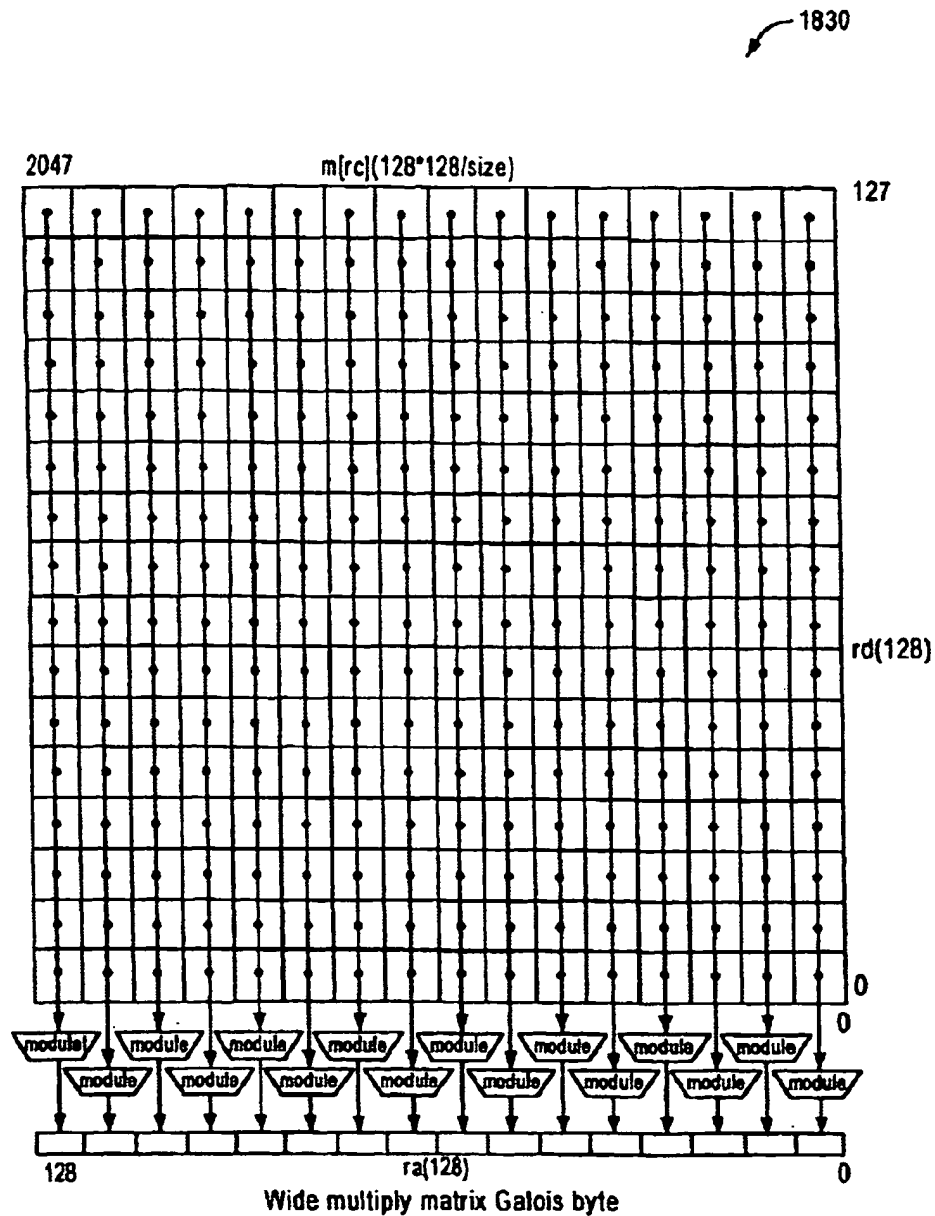


FIG. 18B

U.S. Patent

Apr. 20, 2004

Sheet 48 of 148

US 6,725,356 B2

Definition

1860

```

def c ← PolyMultiply(size,a,b) as
  p[0] ← 02*size
  for k ← 0 to size-1
    p[k+1] ← p[k] ^ ak ? (0size-k || b || 0k) : 02*size
  endfor
  c ← p[size]
enddef

def c ← PolyResidue(size,a,b) as
  p[0] ← a
  for k ← size-1 to 0 by -1
    p[k-1] ← p[k] ^ p[0]size+k ? (0size-k || 1 || b || 0k) : 02*size
  endfor
  c ← p[size]size-1..0
enddef

def WideMultiplyMatrixGalois(op,gsize,rd,rc,rb,ra)
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 64)
  b ← RegRead(rb, 128)
  lsize ← log(gsize)
  if clsize-4..0 ≠ 0 then
    raise AccessDisallowedByVirtualAddress
  endif
  if c3..lsize-3 ≠ 0 then
    wsize ← (c and (0-c)) || 04
    t ← c and (c-1)
  else
    wsize ← 128
    t ← c
  endif
  lsize ← log(wsize)
  if twsize+6-lsize..lsize-3 ≠ 0 then
    msize ← (t and (0-t)) || 04
    VirtAddr ← t and (t-1)
  else
    msize ← 128*wsize/gsize
    VirtAddr ← t
  endif
  case op of
    W.MUL.MAT.G.8.B:
      order ← B
    W.MUL.MAT.G.8.L:
      order ← L
  endcase

```

FIG. 18C-1

U.S. Patent

Apr. 20, 2004

Sheet 49 of 148

US 6,725,356 B2

1860

```

m ← LoadMemory(c, VirtAddr, msize, order)
for i ← 0 wsize-gsize by gsize
  q[0] ← 02*gsize
  for j ← 0 to vsize-gsize by gsize
    k ← i+wsize*j0..lgsize
    q[j+gsize] ← q[j] ^ PolyMultiply(gsize, mk+gsize-1..k, dj+gsize-1..j)
  endfor
  agsize-1+i..i ← PolyResidue(gsize, q[vsize], bgsize-1..0)
endfor
a127..wsize ← 0
RegWrite(ra, 128, a)
enddef

```


FIG. 18C-2

U.S. Patent

Apr. 20, 2004

Sheet 50 of 148

US 6,725,356 B2

1890


Exceptions

Access disallowed by virtual address

Access disallowed by tag

Access disallowed by global TB

Access disallowed by local TB

Access detail required by tag

Access detail required by local TB

Access detail required by global TB

Local TB miss

Global TB miss

FIG. 18D

U.S. Patent

Apr. 20, 2004

Sheet 51 of 148

US 6,725,356 B2

1910

Operation codes

E.MUL.ADD.X	Ensemble multiply add extract
E.CON.X	Ensemble convolve extract

Format

E.op rd@rc,rb,ra

rd=gop(rd,rc,rb,ra)

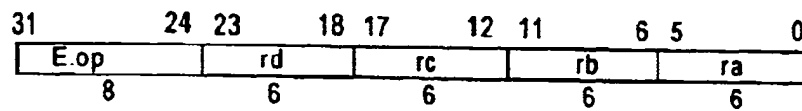


FIG. 19A

U.S. Patent

Apr. 20, 2004

Sheet 52 of 148

US 6,725,356 B2

1910

Figures 19B and 20B has blank fields: should be.

fsize	dpos	x	s	n	m	l	rnd	gssp
-------	------	---	---	---	---	---	-----	------

FIG. 19B

U.S. Patent

Apr. 20, 2004

Sheet 53 of 148

US 6,725,356 B2

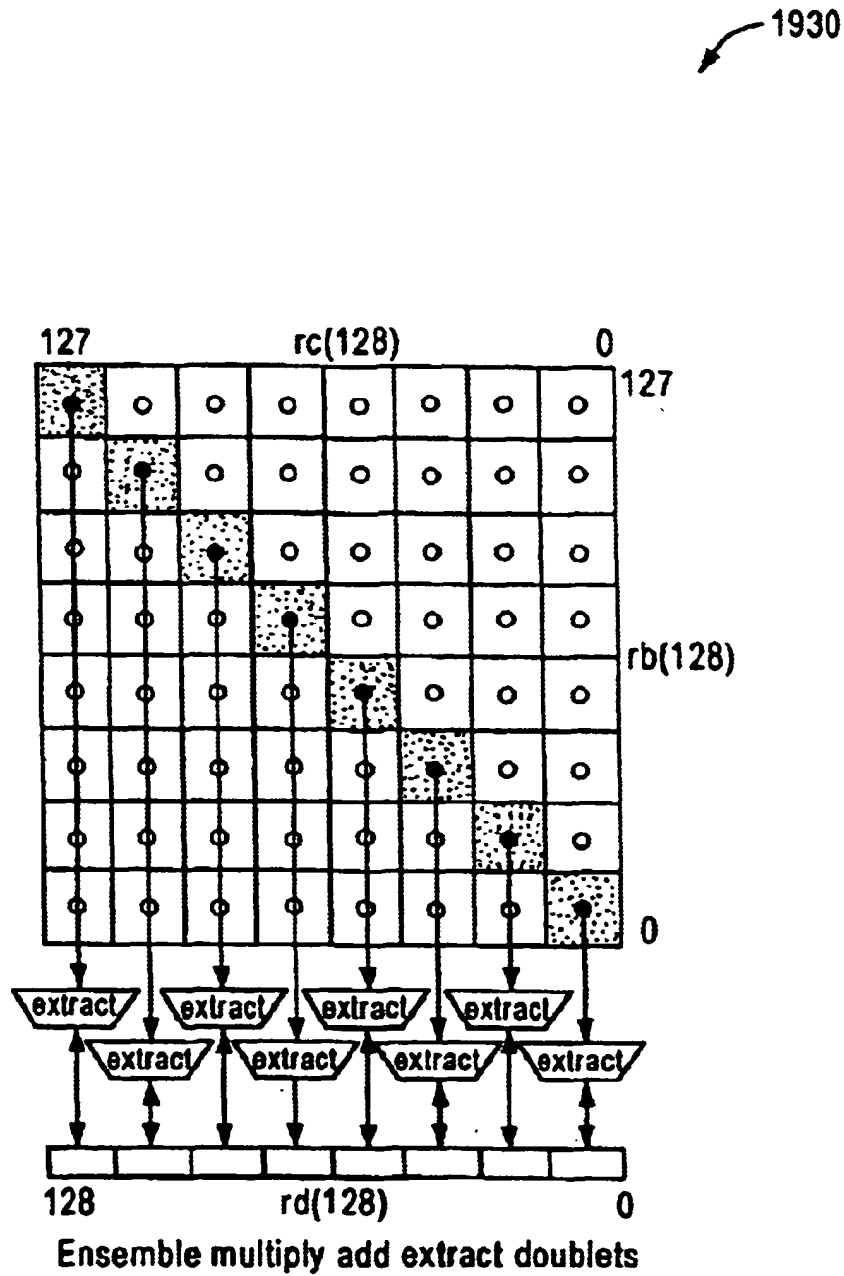


FIG. 19C

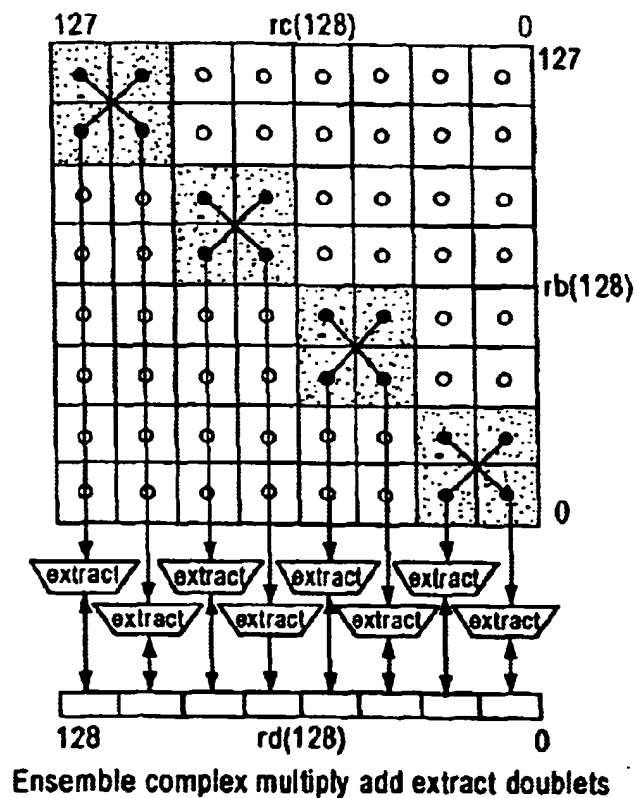
U.S. Patent

Apr. 20, 2004

Sheet 54 of 148

US 6,725,356 B2

1945



This ensemble-multiply-add-extract instructions (E.MUL.ADD.X), when the x bit is set, multiply the low-order 64 bits of each of the rc and rb registers and produce extended (double-size) results.

FIG. 19D

U.S. Patent

Apr. 20, 2004

Sheet 55 of 148

US 6,725,356 B2

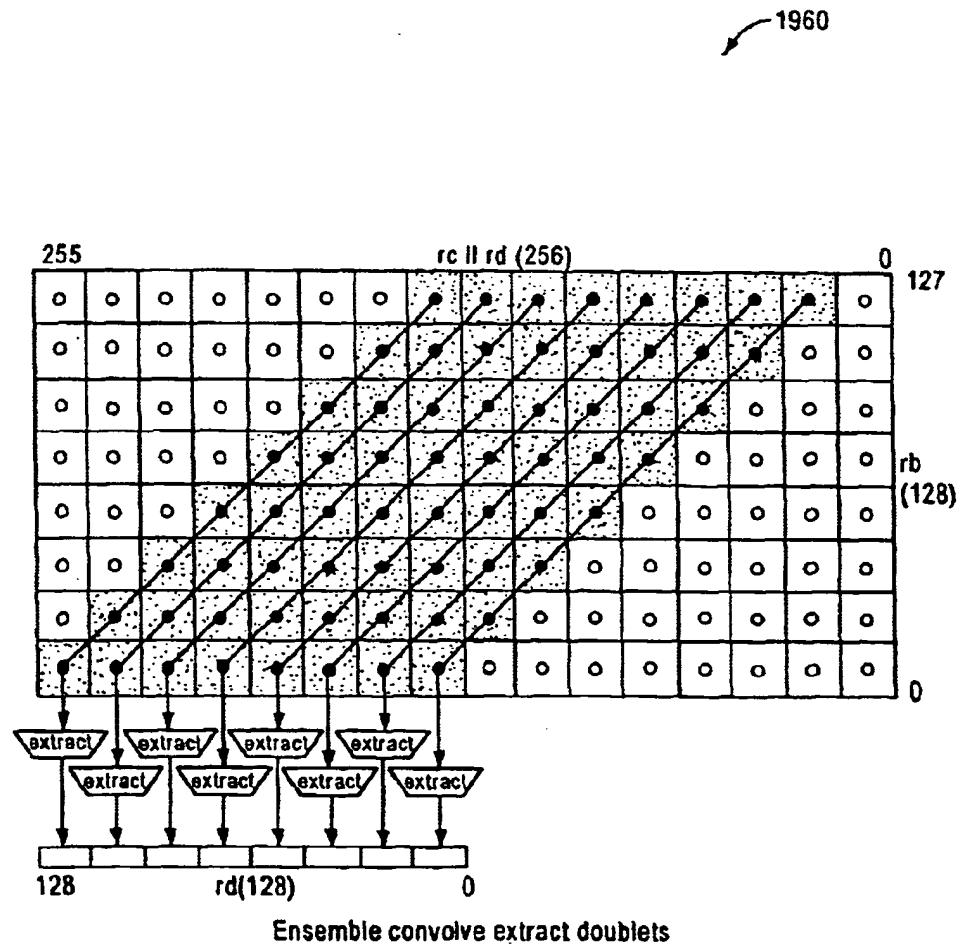


FIG. 19E

U.S. Patent

Apr. 20, 2004

Sheet 56 of 148

US 6,725,356 B2

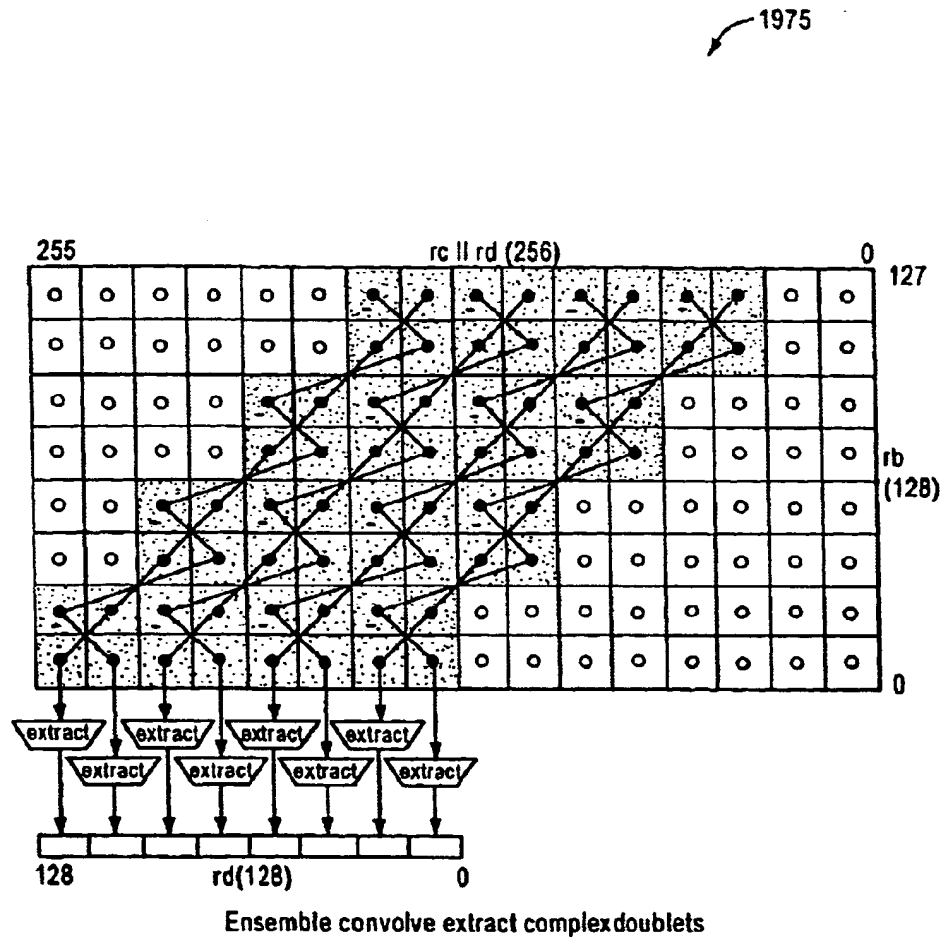


FIG. 19F

U.S. Patent

Apr. 20, 2004

Sheet 57 of 148

US 6,725,356 B2

Definition

1990

```

def mul(size,h,vs,v,i,ws,w,j) as
  mul ← ((vs&vsize-1+i)h-size||vsize-1+i..i) * ((ws&wsize-1+j)h-size||wsize-1+j..j)
enddef

def EnsembleExtractInplace(op,ra,rb,rc,rd) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case b8..0 of
    0..255:
      sgsz ← 128
    256..383:
      sgsz ← 64
    384..447:
      sgsz ← 32
    448..479:
      sgsz ← 16
    480..495:
      sgsz ← 8
    496..503:
      sgsz ← 4
    504..507:
      sgsz ← 2
    508..511:
      sgsz ← 1
  endcase
  l ← a11
  m ← a12
  n ← a13
  signed ← a14
  x ← a15
  case op of
    E.CON.X:
      if (sgsz < 8) then
        gsz ← 8
      elseif (sgsz*(n-1)*(x+1) > 128 then
        gsz ← 128/(n-1)/(x+1)
      else
        gsz ← sgsz
      endif
      lgsize ← log(gsz)
      wsize ← 128/(x+1)
  endcase

```

FIG. 19G-1

U.S. Patent

Apr. 20, 2004

Sheet 58 of 148

US 6,725,356 B2

vsize \leftarrow 128
 ds \leftarrow cs \leftarrow signed
 bs \leftarrow signed \wedge m
 zs \leftarrow signed or m or n
 zsize \leftarrow gsize*(x+1)
 h \leftarrow (2*gszsize) + log(vsize) - lgszsize
 spos \leftarrow (a_{8..0}) and (2*gszsize-1)

1990

E.MUL.ADD.X:

if(sgszsize < 9) then
 gszsize \leftarrow 8
 elseif (sgszsize*(n+1)*(x+1) > 128) then
 gszsize \leftarrow 128/(n+1)/(x+1)
 else
 gszsize \leftarrow sgszsize
 endif
 ds \leftarrow signed
 cs \leftarrow signed \wedge m
 zs \leftarrow signed or m or n
 zsize \leftarrow gszsize*(x+1)
 h \leftarrow (2*gszsize) + n
 spos \leftarrow (a_{8..0}) and (2*gsszsize-1)
 endcase
 dpos \leftarrow (0|| a_{23..16}) and (zsize-1)
 r \leftarrow spos
 sfszsize \leftarrow (0|| a_{31..24}) and (zsize-1)
 tfszsize \leftarrow (sfszsize = 0) or ((sfszsize+dpos) > zsize) ? zsize-dpos : sfszsize
 fsize \leftarrow (tfszsize + spos > h) ? h - spos : tfszsize
 if (b_{10..9} = Z) and not as then
 rnd \leftarrow F
 else
 rnd \leftarrow b_{10..9}
 endif

FIG. 19G-2

U.S. Patent

Apr. 20, 2004

Sheet 59 of 148

US 6,725,356 B2

1990

```

for k ← 0 to wsize-zsize by zsize
  i ← k*gsize/zsize
  case op of
    E.CON.X:
      q[0] ← 0
      for j ← 0 to vsize-gsize by gsize
        if n then
          if (~) & j & gsize = 0 then
            q[j+gsize] ← q[j] + mul(gsize,h,ms,m,i+
              128-j,bs,b,j)
          else
            q[j+gsize] ← q[j] - mul(gsize,h,ms,m,i+
              128-j+2*gsize,bs,b,j)
          endif
        else
          q[j+gsize] ← q[j] + mul(gsize,h,ms,m,i+
            128-j,bs,b,j)
        endif
      endfor
      p ← q[vsize]
    E.MUL.ADD.X:
      di ← ((ds and dk+zsize-1)h-zsize-r || (dk+zsize-1..k) || 0')
      if n then
        if (i and gsize) = 0 then
          p ← mul(gsize,h,ds,d,i,cs,c,i)-
mul(gsize,h,ds,d,i+gsize,cs,c,i+gsize)+di
        else
          p ← mul(gsize,h,ds,d,i,cs,c,i+gsize)+mul(gsize,h,ds,d,i,cs,c,i+gsize)+di
        endif
      else
        p ← mul(gsize,h,ds,d,i,cs,c,i) + di
      endif
    endif
  endcase

```

FIG. 19G-3

U.S. Patent

Apr. 20, 2004

Sheet 60 of 148

US 6,725,356 B2

1990

```

case rnd of
N:
   $s \leftarrow 0^{h-r} \parallel \sim p_r \parallel p_{r-1}^r$ 
Z:
   $s \leftarrow 0^{h-r} \parallel p_{h-1}^r$ 
F:
   $s \leftarrow 0^h$ 
C:
   $s \leftarrow 0^{h-r} \parallel 1^r$ 
endcase
 $v \leftarrow ((zs \& p_{h-1}) \parallel p) + (0 \parallel s)$ 
if ( $v_{h..r+fsz} = (zs \& v_{r+fsz-1})^{h+1-r-fsz}$ ) or not (l and (op =
EXTRACT)) then
   $w \leftarrow (zs \& v_{r+fsz-1})^{zsize-fsize-dpos} \parallel v_{fsz-1+r..r} \parallel 0^{dpos}$ 
else
   $w \leftarrow (zs ? (v_h) \parallel \sim v_{zsize-dpos-1}^h) : 1^{zsize-dpos} \parallel 0^{dpos}$ 
endif
 $z_{size-1\_k..k} \leftarrow w$ 
endfor
RegWrite(rd, 128, z)
enddef

```

FIG. 19G-4

U.S. Patent

Apr. 20, 2004

Sheet 61 of 148

US 6,725,356 B2

2010

Operation codes

E.MUL.X	Ensemble multiply extract
E.EXTRACT	Ensemble extract
E.SCAL.ADD.X	Ensemble scale and extract

Format

E.op ra=rd,rc,rb

ra=eop(rd,rc,rb)

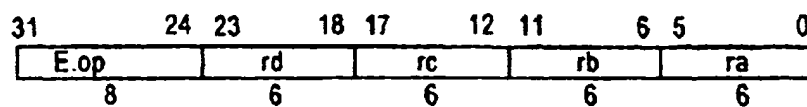


FIG. 20A

U.S. Patent

Apr. 20, 2004

Sheet 62 of 148

US 6,725,356 B2

2015
↙

Figures 19B and 20B has blank fields: should be.

fsize	dpos	x	s	n	m	l	rnd	gssp
-------	------	---	---	---	---	---	-----	------

FIG. 20B

U.S. Patent

Apr. 20, 2004

Sheet 63 of 148

US 6,725,356 B2

2020

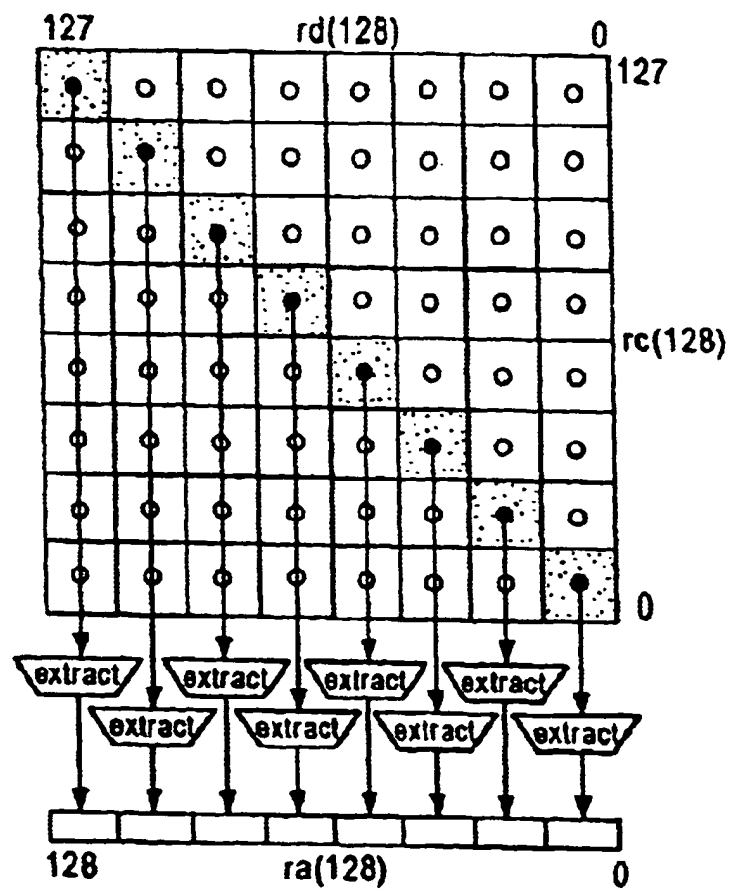


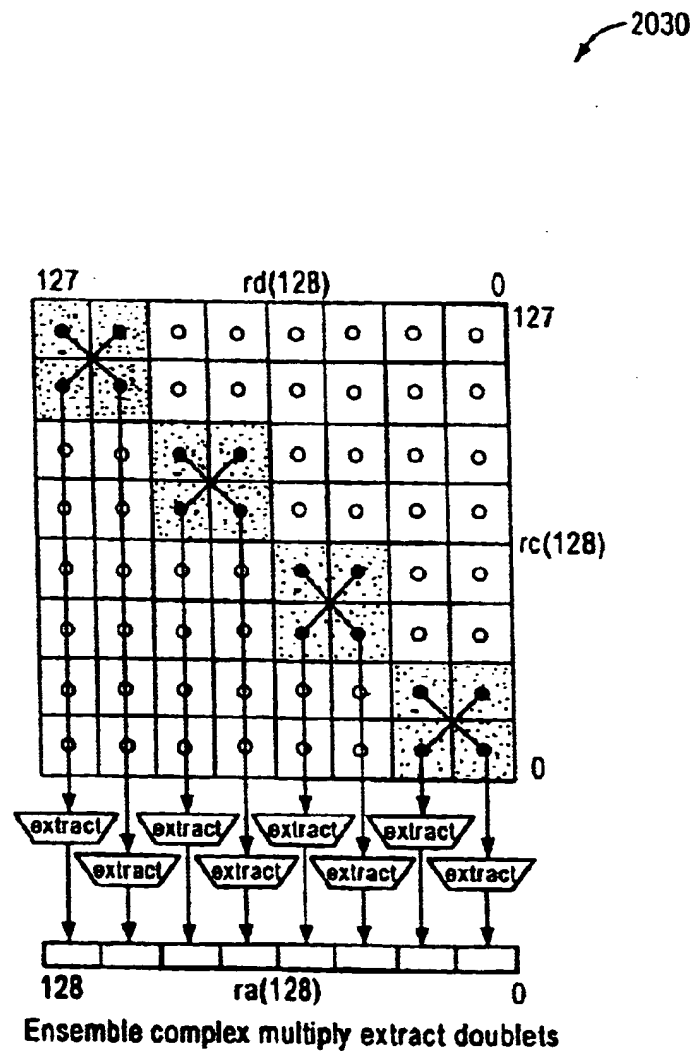
FIG. 20C

U.S. Patent

Apr. 20, 2004

Sheet 64 of 148

US 6,725,356 B2



This ensemble-multiply-extract instructions (E.MUL.X), when the x bit is set, multiply the low-order 64 bits of each of the rc and rb registers and produce extended (double-size) results.

FIG. 20D

U.S. Patent

Apr. 20, 2004

Sheet 65 of 148

US 6,725,356 B2

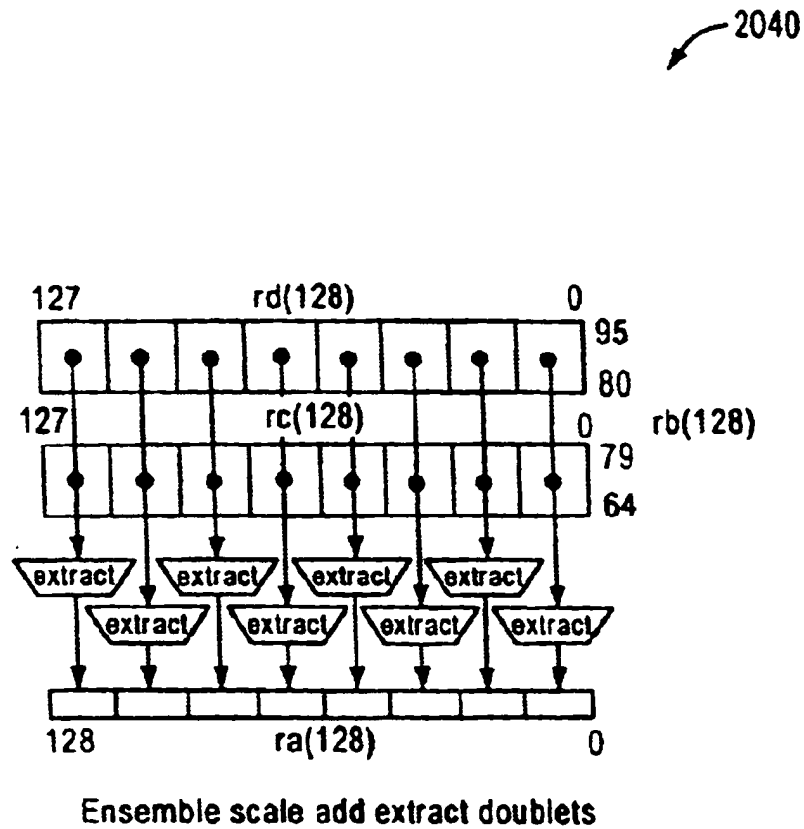


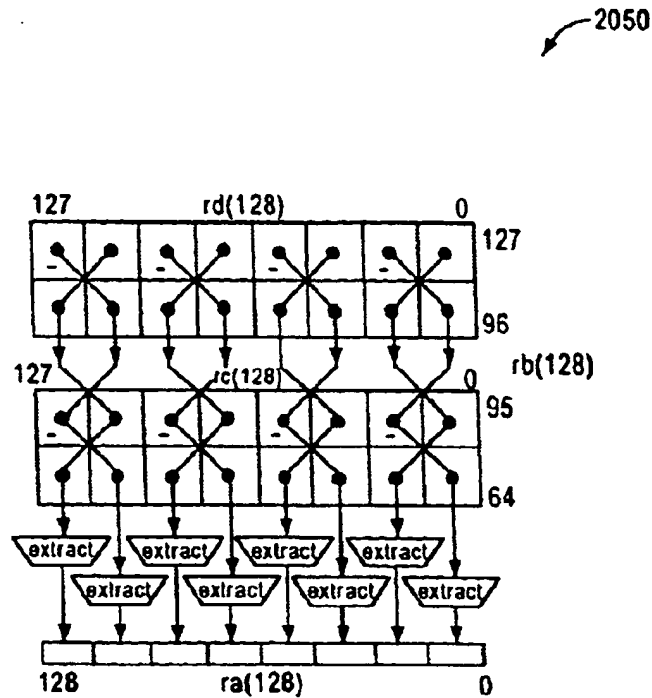
FIG. 20E

U.S. Patent

Apr. 20, 2004

Sheet 66 of 148

US 6,725,356 B2



Ensemble complex scale add extract doublets

The ensemble-scale-add-extract instructions (E.SCLADD.X), when the x bit is set, multiply the low-order 64 bits of each of the rd and re registers by the rb register fields and produce extended (double-size) results.

FIG. 20F

U.S. Patent

Apr. 20, 2004

Sheet 67 of 148

US 6,725,356 B2

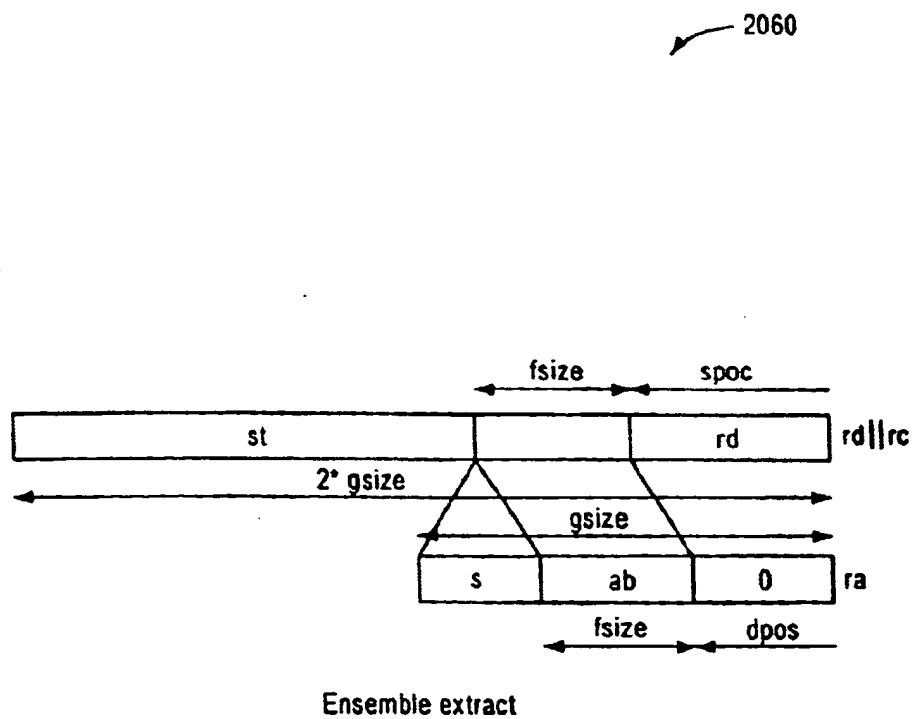


FIG. 20G

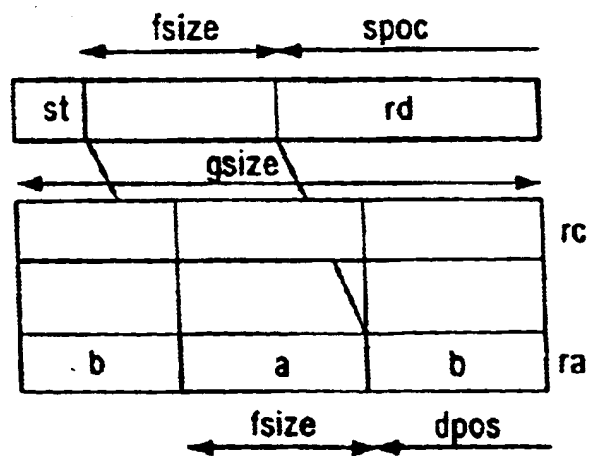
U.S. Patent

Apr. 20, 2004

Sheet 68 of 148

US 6,725,356 B2

2070



Ensemble merge extract

FIG. 20H

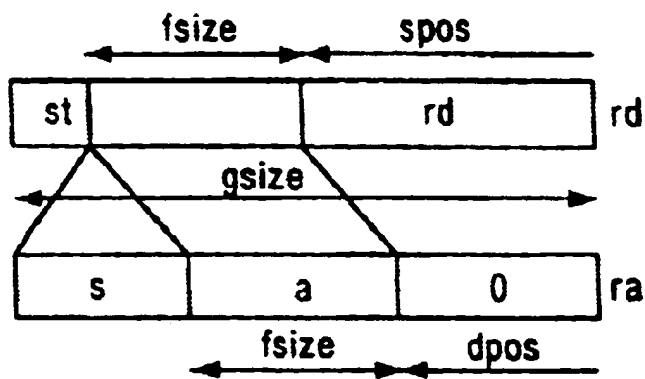
U.S. Patent

Apr. 20, 2004

Sheet 69 of 148

US 6,725,356 B2

2080



Ensemble expand extract

FIG. 20I

U.S. Patent

Apr. 20, 2004

Sheet 70 of 148

US 6,725,356 B2

Definition

```

def mul(size,h,vs,v,i,ws,w,j) as
    mul ← ((vs&vsize-1+i)h-size||vsize-1+i..i) * ((ws&wsize-1+j)h-size||wsize-1+j..j)
enddef

```

2090

```

def EnsembleExtract(op,ra,rb,rc,rd) as
    d ← RegRead(rd, 128)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case ba..0 of
        0..255:
            ssize ← 128
        256..383:
            ssize ← 64
        384..447:
            ssize ← 32
        448..479:
            ssize ← 16
        480..495:
            ssize ← 8
        496..503:
            ssize ← 4
        504..507:
            ssize ← 2
        508..511:
            ssize ← 1
    endcase
    l ← b11
    m ← b12
    n ← b13
    signed ← b14
    x ← b15
    case op of
        E.EXTRACT:
            gsize ← ssize*2(2-(m or x))
            zsize ← ssize
            h ← gsize
            as ← signed
            spos ← (ba..0) and (gsize-1)

```

FIG. 20J-1

U.S. Patent

Apr. 20, 2004

Sheet 71 of 148

US 6,725,356 B2

E.SCAL.ADD.X:

2090

```

if (sgsize < 8) then
  gsize ← 8
elseif (sgsize*(n+1) > 32) then
  gsize ← 32/(n+1)
else
  gsize ← sgsz
endif
ds ← cs ← signed
bs ← signed ^ m
as ← signed or m or n
zsize ← gsize*(x+1)
h ← (2*gsz) + 1 + n
spos ← (b8..0) and (2*gsz-1)

```

E.MUL.X:

```

if (sgsize < 8) then
  gsize ← 8
elseif (sgsize*(n+1)*(x+1) > 128) then
  gsize ← 128/(n+1)/(x+1)
else
  gsize ← sgsz
endif
ds ← signed
cs ← signed ^ m
as ← signed or m or n
zsize ← gsize*(x+1)
h ← (2*gsz) + n
spos ← (b8..0) and (2*gsz-1)

```

endcase

dpos ← (0 || b_{23..16}) and (zsize-1)

r ← spos

sfsz ← (0 || b_{31..24}) and (zsize-1)

tfsz ← (sfsz = 0) or ((sfsz+dpos) > zsize) ? zsize-dpos : sfsz

fsz ← (tfsz + spos > h) ? h - spos : tfsz

if (b_{10..9}=Z) and not as then

rnd ← F

else

rnd ← b

endif

FIG. 20J-2

U.S. Patent

Apr. 20, 2004

Sheet 72 of 148

US 6,725,356 B2

2090

```

for j ← 0 to 128-zsize by zsize
  i ← j*gszsize/zsize
  case op of
    E.EXTRACT:
      if m or x then
        p ← dgszsize+i-1..i
      else
        p ← (d||c)gszsize+i-1..i
      endif
    E.MUL.X:
      if n then
        if (i and gsize) = 0 then
          p ← mul(gsize,h,ds,d,i,cs,c,i)-
mul(gsize,h,ds,d,i+gszsize,cs,c,i+gszsize)
        else
          p ←
mul(gsize,h,ds,d,i,cs,c,i+gszsize)+mul(gsize,h,ds,d,i,cs,c,i+gszsize)
        endif
      else
        p ← mul(gsize,h,ds,d,i,cs,c,i)
      endif
    E.SCAL.ADD.X:
      if n then
        if (i and gsize) = 0 then
          p ← mul(gsize,h,ds,d,i,bs,b,64+2*gszsize)
            + mul(gsize,h,cs,c,i,bs,b,64)
            - mul(gsize,h,ds,d,i+gszsize,bs,b,64+3*gszsize)
            - mul(gsize,h,cs,c,i+gszsize,bs,b,64+gszsize)
        else
          p ← mul(gsize,h,ds,d,i,bs,b,64+3*gszsize)
            + mul(gsize,h,cs,c,i,bs,b,64+gszsize)
            + mul(gsize,h,ds,d,i+gszsize,bs,b,64+2*gszsize)
            + mul(gsize,h,cs,c,i+gszsize,bs,b,64)
        endif
      else
        p ← mul(gsize,h,ds,d,i,bs,b,64+gszsize) + mul(gsize
            ,h,cs,c,i,bs,b,64)
      endif
    endif
  endcase

```

FIG. 20J-3

U.S. Patent

Apr. 20, 2004

Sheet 73 of 148

US 6,725,356 B2

2090

```

case rnd of
  N:
     $s \leftarrow 0^{h-r} || \sim p_r || p_r^{r-1}$ 
  Z:
     $s \leftarrow 0^{h-r} || p_{h-1}^r$ 
  F:
     $s \leftarrow 0^h$ 
  C:
     $s \leftarrow 0^{h-r} || 1^r$ 
endcase
 $v \leftarrow ((as \& p_{h-1}) || p) + (0 || s)$ 
if  $(v_{h..r+fsz} = (as \& v_{r+fsz-1})^{h+1-r-fsz})$  or not (1 and (op =
  E.EXTRACT)) then
   $w \leftarrow (as \& v_{r+fsz-1})^{zsize-fsize-dpos} || v_{fsz-1+r..r} || 0^{dpos}$ 
else
   $w \leftarrow (s ? (v_h || \sim v_h^{zsize-dpos-1}) : 1^{zsize-dpos}) || 0^{dpos}$ 
endif
if m and (op = E.EXTRACT) then
   $z_{size-1+j..j} \leftarrow c_{size-1+j..dpos+fsz+j} || w_{dpos+fsz-1..dpos} ||$ 
     $c_{dpos-1+j..j}$ 
else
   $z_{size-1+j..j} \leftarrow w$ 
endif
endfor
RegWrite(ra, 128, z)
enddef

```

FIG. 20J-4

U.S. Patent

Apr. 20, 2004

Sheet 74 of 148

US 6,725,356 B2

2110

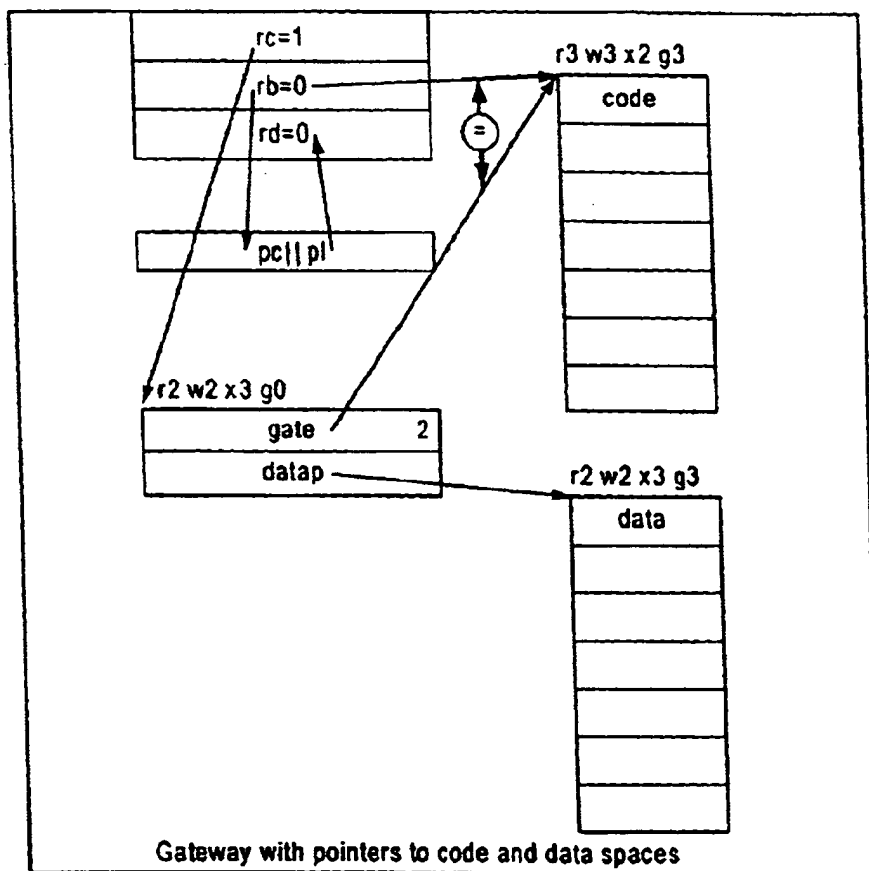


FIG. 21A

U.S. Patent

Apr. 20, 2004

Sheet 75 of 148

US 6,725,356 B2

2130

Typical dynamic-linked, inter-gateway calling sequence:

caller:

caller	AA.DDI	sp@-size	// allocate caller stack frame
	S.I.64.A	lp,sp,off	
	S.I.64.A	dp,sp,off	
	...		
	L.I.64.A	lp=dp,off	// load lp
	L.I.64.A	dp=dp,off	// load dp
	B.GATE		
	L.I.64.A	dp,sp,off	
	...(code using dp)		
	L.I.64.A	lp=sp,off	// restore original lp register
	A.ADDI	sp=size	// deallocate caller stack frame
	B	lp	// return

callee (non-leaf):

callee:	L.I.64.A	dp=dp,off	// load dp with data pointer
	S.I.64.A	sp,dp,off	
	L.I.64.A	sp=dp,off	// new stack pointer
	S.I.64.A	lp,sp,off	
	S.I.64.A	dp,sp,off	
	...(using dp)		
	L.I.64.A	dp,sp,off	
	...(code using dp)		
	L.I.64.A	lp=sp,off	// restore original lp register
	L.I.64.A	sp=sp,off	// restore original sp register
	B.DOWN	lp	

callee (leaf, no stack):

callee:	...(using dp)	
	B.DOWN	lp

FIG. 21B

U.S. Patent

Apr. 20, 2004

Sheet 76 of 148

US 6,725,356 B2

2160

Operation codes

B.GATE	Branch gateway
--------	----------------

Equivalencies

B.GATE	← B.GATE 0
--------	------------

Format

B.GATE rb

bgate(rb)

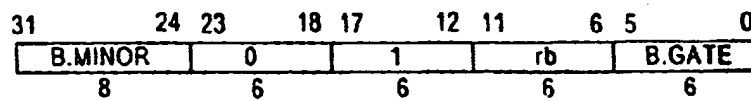


FIG. 21C

U.S. Patent

Apr. 20, 2004

Sheet 77 of 148

US 6,725,356 B2

2170

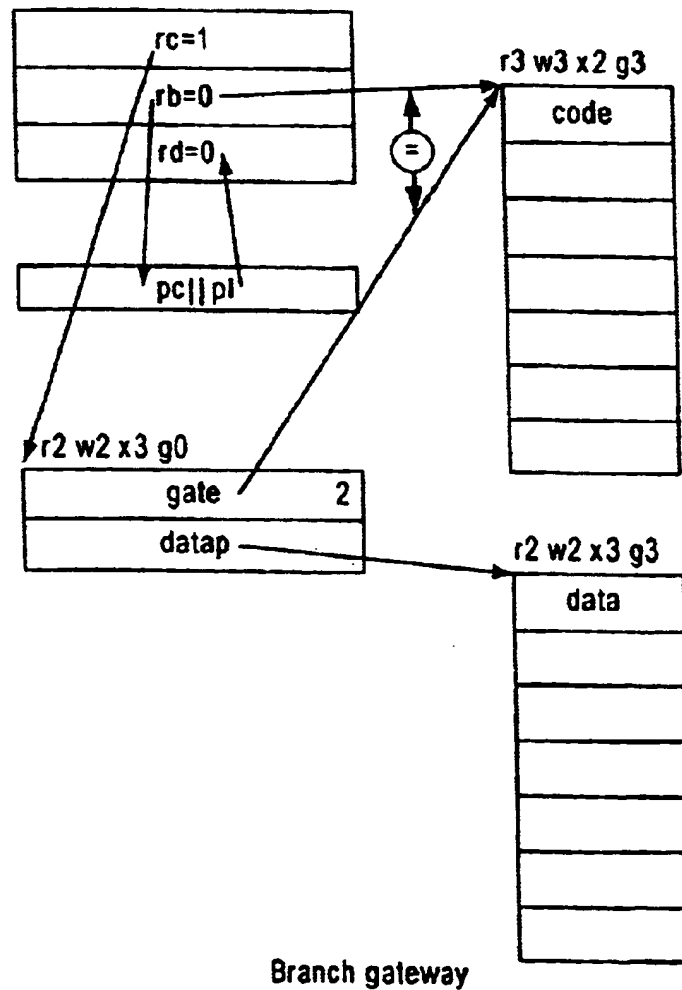


FIG. 21D

U.S. Patent

Apr. 20, 2004

Sheet 78 of 148

US 6,725,356 B2

2190

Definition

```

def BranchGateway(rd,rc,rb) as
  c ← RegRead(rc, 64)
  b ← RegRead(rb, 64)
  if (rd ≠ 0) or (rc ≠ 1) then
    raise ReservedInstruction
  endif
  if c2..0 ≠ 0 then
    raise AccessDisallowedByVirtualAddress
  endif
  d ← ProgramCounter63..2+1 || PrivilegeLevel
  if PrivilegeLevel < b1..0 then
    m ← LoadMemoryG(c,c,64,L)
    if b ≠ m then
      raise GatewayDisallowed
    endif
    PrivilegeLevel ← b1..0
  endif
  ProgramCounter ← b63..2 || 02
  RegWrite(rd, 64, d)
  raise TakenBranch
enddef

```


FIG. 21E

U.S. Patent

Apr. 20, 2004

Sheet 79 of 148

US 6,725,356 B2

2199


Exceptions

Reserved Instruction
Gateway disallowed
Access disallowed by virtual address
Access disallowed by tag
Access disallowed by global TB
Access disallowed by local TB
Access detail required by tag
Access detail required by local TB
Access detail required by global TB
Local TB miss
Global TB miss

FIG. 21F

U.S. Patent

Apr. 20, 2004

Sheet 80 of 148

US 6,725,356 B2

2210

Operation codes

E.SCAL.ADD.F.16	Ensemble scale add floating-point half
E.SCAL.ADD.F.32	Ensemble scale add floating-point single
E.SCAL.ADD.F.64	Ensemble scale add floating-point double

Selection

class	op	prec		
scale add	E.SCAL.ADD.F	16	32	64

Format

E.op.prec ra=rd,rc,rb

ra=eopprec(rd,rc,rb)

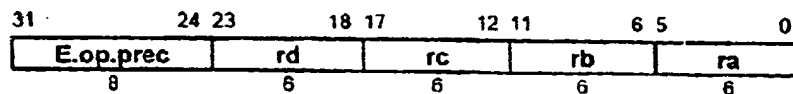


FIG. 22A

U.S. Patent

Apr. 20, 2004

Sheet 81 of 148

US 6,725,356 B2

2230

Definition

```
def EnsembleFloatingPointTernary(op,prec,rd,rc,rb,ra) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  for i ← 0 to 128-prec by prec
    di ← F(prec,d[i+prec-1..i])
    ci ← F(prec,c[i+prec-1..i])
    ai ← fadd(fmul(di, F(prec,b[prec-1..0])), fmul(ci, F(prec,b[2*prec-1..prec])))
    ai+prec-1..i ← PackF(prec, ai, none)
  endfor
  RegWrite(ra, 128, a)
enddef
```

FIG. 22B

U.S. Patent

Apr. 20, 2004

Sheet 82 of 148

US 6,725,356 B2

2310

Operation codes

G.BOOLEAN	Group boolean
-----------	---------------

Selection

operation	function (binary)	function (decimal)
d	11110000	240
c	11001100	204
b	10101010	176
d&c&b	10000000	128
(d&c) b	11101010	234
d c b	11111110	254
d?c:b	11001010	202
d^c^b	10010110	150
~d^c^b	01101001	105
0	00000000	0

Format

G.BOOLEAN rd@trc,rb,f

rd=gbooleani(rd,rc,rb,f)

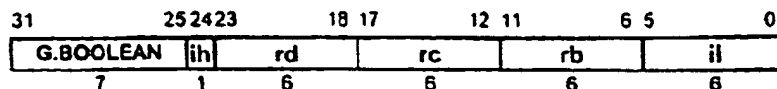


FIG. 23A

U.S. Patent

Apr. 20, 2004

Sheet 83 of 148

US 6,725,356 B2

2320

```

if f6=f5 then
  if f2=f1 then
    if f2 then
      rc ← max(trc, trb)
      rb ← min(trc, trb)
    else
      rc ← min(trc, trb)
      rb ← max(trc, trb)
    endif
    ih ← 0
    il ← 0 || f6 || f7 || f4 || f3 || f0
  else
    if f2 then
      rc ← trb
      rb ← trc
    else
      rc ← trc
      rb ← trb
    endif
    ih ← 0
    il ← 1 || f6 || f7 || f4 || f3 || f0
  endif
else
  ih ← 1
  if f6 then
    rc ← trb
    rb ← trc
    il ← f1 || f2 || f7 || f4 || f3 || f0
  else
    rc ← trc
    rb ← trb
    il ← f2 || f1 || f7 || f4 || f3 || f0
  endif
endif

```


FIG. 23B

U.S. Patent

Apr. 20, 2004

Sheet 84 of 148

US 6,725,356 B2

 2330
Definition

```

def GroupBoolean (ih,rd,rc,rb,il)
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  if ih=0 then
    if il5=0 then
      f ← il3 || il4 || il4 || il2 || il1 || (rc>rb)2 || il0
    else
      f ← il3 || il4 || il4 || il2 || il1 || 0 || 1 || il0
    endif
  else
    f ← il3 || 0 || 1 || il2 || il1 || il5 || il4 || il0
  endif
  for i ← 0 to 127 by size
    ai ← f(di || ci || bi)
  endfor
  RegWrite(rd, 128, a)
enddef

```

FIG. 23C

U.S. Patent

Apr. 20, 2004

Sheet 85 of 148

US 6,725,356 B2

2410

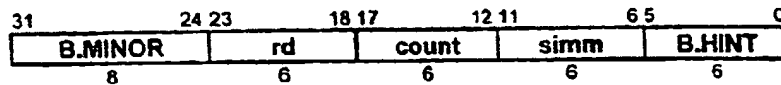
Operation codes

B.HINT	Branch Hint
--------	-------------

Format

B.HINT badd,count,rd

bhint(badd,count,rd)



simm ← badd-pc-4

FIG. 24A

U.S. Patent

Apr. 20, 2004

Sheet 86 of 148

US 6,725,356 B2

↖ 2430

Definition

```
def BranchHint(rd,count,simm) as
  d ← RegRead(rd, 64)
  if (d1..0) ≠ 0 then
    raise AccessDisallowedByVirtualAddress
  endif
  FetchHint(ProgramCounter + 4 + (0 || simm || 02), d63..2 || 02, count)
enddef
```


FIG. 24B

U.S. Patent

Apr. 20, 2004

Sheet 87 of 148

US 6,725,356 B2

 **2460**

Exceptions

Access disallowed by virtual address

FIG. 24C

U.S. Patent

Apr. 20, 2004

Sheet 88 of 148

US 6,725,356 B2

2510

Operation codes

E.SINK.F.16	Ensemble convert floating-point doublets from half nearest default
E.SINK.F.16C	Ensemble convert floating-point doublets from half ceiling
E.SINK.F.16.C.D	Ensemble convert floating-point doublets from half ceiling default
E.SINK.F.16.F	Ensemble convert floating-point doublets from half floor
E.SINK.F.16.F.D	Ensemble convert floating-point doublets from half floor default
E.SINK.F.16.N	Ensemble convert floating-point doublets from half nearest
E.SINK.F.16.X	Ensemble convert floating-point doublets from half exact
E.SINK.F.16.Z	Ensemble convert floating-point doublets from half zero
E.SINK.F.16.Z.D	Ensemble convert floating-point doublets from half zero default
E.SINK.F.32	Ensemble convert floating-point quadlets from single nearest default
E.SINK.F.32.C	Ensemble convert floating-point quadlets from single ceiling
E.SINK.F.32.C.D	Ensemble convert floating-point quadlets from single ceiling default
E.SINK.F.32.F	Ensemble convert floating-point quadlets from single floor
E.SINK.F.32.F.D	Ensemble convert floating-point quadlets from single floor default
E.SINK.F.32.N	Ensemble convert floating-point quadlets from single nearest
E.SINK.F.32.X	Ensemble convert floating-point quadlets from single exact
E.SINK.F.32.Z	Ensemble convert floating-point quadlets from single zero
E.SINK.F.32.Z.D	Ensemble convert floating-point quadlets from single zero default
E.SINK.F.64	Ensemble convert floating-point octlets from double nearest default
E.SINK.F.64.C	Ensemble convert floating-point octlets from double ceiling
E.SINK.F.64.C.D	Ensemble convert floating-point octlets from double ceiling default
E.SINK.F.64.F	Ensemble convert floating-point octlets from double floor
E.SINK.F.64.F.D	Ensemble convert floating-point octlets from double floor default
E.SINK.F.64.N	Ensemble convert floating-point octlets from double nearest
E.SINK.F.64.X	Ensemble convert floating-point octlets from double exact
E.SINK.F.64.Z	Ensemble convert floating-point octlets from double zero
E.SINK.F.64.Z.D	Ensemble convert floating-point octlets from double zero default
E.SINK.F.128	Ensemble convert floating-point hexlet from quad nearest default
E.SINK.F.128.C	Ensemble convert floating-point hexlet from quad ceiling
E.SINK.F.128.C.D	Ensemble convert floating-point hexlet from quad ceiling default
E.SINK.F.128.F	Ensemble convert floating-point hexlet from quad floor
E.SINK.F.128.F.D	Ensemble convert floating-point hexlet from quad floor default
E.SINK.F.128.N	Ensemble convert floating-point hexlet from quad nearest
E.SINK.F.128.X	Ensemble convert floating-point hexlet from quad exact
E.SINK.F.128.Z	Ensemble convert floating-point hexlet from quad zero
E.SINK.F.128.Z.D	Ensemble convert floating-point hexlet from quad zero default

FIG. 25A-1

U.S. Patent

Apr. 20, 2004

Sheet 89 of 148

US 6,725,356 B2

2510

Selection

	op	prec	round/trap
integer from float	SINK	16 32 64 128	NONE C F N X Z C.D F.D Z.D

Format

E.SINK.F.prec.rnd rd=rc

rd=esinkfprecrnd(rc)

31	24 23	18 17	12 11	6 5	0
E.prec	rd	rc	E.SINK.F.rnd	E.UNARY	
8	6	6	6	6	

FIG. 25A-2

U.S. Patent

Apr. 20, 2004

Sheet 90 of 148

US 6,725,356 B2

2530

Definition

```
def EnsembleSinkFloatingPoint(prec,round,rd,rc) as
  c ← RegRead(rc, 128)
  for i ← 0 to 128-prec by prec
    ci ← F(prec,ci+prec-1..i)
    ai+prec-1..i ← fsinkr(prec, ci, round)
  endfor
  RegWrite[rd, 128, a]
enddef
```


FIG. 25B

U.S. Patent

Apr. 20, 2004

Sheet 91 of 148

US 6,725,356 B2

2560


Exceptions

Floating-point arithmetic

FIG. 25C

U.S. Patent

Apr. 20, 2004

Sheet 92 of 148

US 6,725,356 B2

Definition 2570

```

def eb ← ebits(prec) as
  case prec of
    16:
      eb ← 5
    32:
      eb ← 8
    64:
      eb ← 11
    128:
      eb ← 15
  endcase
enddef

def eb ← ebias(prec) as
  eb ← 0 || 1ebits(prec)-1
enddef

def fb ← fbits(prec) as
  fb ← prec - 1 - eb
enddef

def a ← F(prec, ai) as
  a.s ← aiprec-1
  ae ← aiprec-2..fbits(prec)
  af ← aifbits(prec)-1..0
  if ae = 1ebits(prec) then
    if af = 0 then
      a.t ← INFINITY
    elseif af fbits(prec)-1 then
      a.t ← SNaN
      a.e ← -fbits(prec)
      a.f ← 1 || af fbits(prec)-1..0
    else
      a.t ← QNaN
      a.e ← -fbits(prec)
      a.f ← af
    endif
  elseif ae = 0 then
    if af = 0 then
      a.t ← ZERO

```

FIG. 25D-1

U.S. Patent

Apr. 20, 2004

Sheet 93 of 148

US 6,725,356 B2

```

else
    a.t ← NORM
    a.e ← 1-ebias(pec)-fbits(pec)
    a.f ← 0|| af
endif
else
    a.t ← NORM
    a.e ← ae-ebias(pec)-fbits(pec)
    a.f ← 1|| af
endif
enddef

def a ← DEFAULTQNaN as
    a.s ← 0
    a.t ← QNaN
    a.e ← -1
    a.f ← 1
endder

def a ← DEFAULTSNAN as
    a.s ← 0
    a.t ← SNAN
    a.e ← -1
    a.f ← 1
enddef
    
```

2570

FIG. 25D-2

U.S. Patent

Apr. 20, 2004

Sheet 94 of 148

US 6,725,356 B2

2570

```
def fadd(a,b) as faddr(a,b,N) endder
```

```
def c ← faddr(a,b,round) as
```

```
  if a.t=NORM and b.t=NORM then
```

```
    // d,e are a,b with exponent aligned and fraction adjusted
```

```
    if a.e > b.e then
```

```
      d ← a
```

```
      e.t ← b.t
```

```
      e.s ← b.s
```

```
      e.e ← a.e
```

```
      e.f ← b.f || 0a.e-b.e
```

```
    else if a.e < b.e then
```

```
      d.t ← a.t
```

```
      d.s ← a.s
```

```
      d.e ← b.e
```

```
      d.f ← a.f || 0b.e-a.e
```

```
      e ← b
```

```
    endif
```

```
    c.t ← d.t
```

```
    c.e ← d.e
```

```
    if d.s = e.s then
```

```
      c.s ← d.s
```

```
      c.f ← d.f + e.f
```

```
    elseif d.f > e.f then
```

```
      c.s ← d.s
```

```
      c.f ← d.f - e.f
```

```
    elseif d.f < e.f then
```

```
      c.s ← e.s
```

```
      c.f ← e.f - d.f
```

```
    else
```

```
      c.s ← r=F
```

```
      c.t ← ZERO
```

```
    endif
```

FIG. 25D-3

U.S. Patent

Apr. 20, 2004

Sheet 95 of 148

US 6,725,356 B2

2570

```

// priority is given to be operand for NaN propagation
elseif (b.t=SNAN) or (b.t=QNAN) then
    c ← b
elseif (a.t=SNAN) or (a.t=QNAN) then
    c ← a
elseif a.t=ZERO and b.t=ZERO then
    c.t ← ZERO
    c.s ← (a.s and b.s) or (round=F and (a.s or b.s))
// NULL values are like zero, but do not combine with ZERO to alter sign
elseif a.t=ZERO or a.t=NULL then
    c ← b
elseif b.t=ZERO or b.t=NULL then
    c ← a
elseif a.t=INFINITY and b.t=INFINITY then
    if a.s ≠ b.s then
        c ← DEFAULTSNAN // Invalid
    else
        c ← a
    endif
elseif a.t=INFINITY then
    c ← a
elseif b.t=INFINITY then
    c ← b
else
    assert FALSE // should have covered all the cases above
endif
enddef

def b ← fneg(a) as
    b.s ← ~a.s
    b.t ← a.t
    b.e ← a.e
    b.f ← a.f
enddef

def fsub(a,b) as fsubr(a,b,N) enddef

def fsubr(a,b,round) as faddr(a,fneg(b),round) enddef

def frsub(a,b) as frsubr(a,b,N) enddef

def frsubr(a,b,round) as faddr(fneg(a),b,round) enddef

```

FIG. 25D-4

U.S. Patent

Apr. 20, 2004

Sheet 96 of 148

US 6,725,356 B2

2570

```

def c ← fcom(a,b) as
  if (a.t=SNAN) or (a.t=QNAN) or (b.t=SNAN) or (b.t=QNAN) then
    c ← U
  elseif a.t=INFINITY and b.t=INFINITY then
    if a.s ≠ b.s then
      c ← (a.s=0) ? G : L
    else
      c ← E
    endif
  elseif a.t=INFINITY then
    c ← (a.s=0) ? G : L
  elseif b.t=INFINITY then
    c ← (b.s=0) ? L
  elseif a.t=NORM and b.t=NORM then
    if a.s ≠ b.s then
      c ← (a.s=0) ? G : L
    else
      if a.e > b.e then
        af ← a.f
        bf ← b.f || 0a.e-b.e
      else
        af ← a.f || 0b.e-a.e
        bf ← b.f
      endif
      if af = bf then
        c ← E
      else
        c ← ((a.s=0) ^ (af > bf)) ? G : L
      endif
    endif
  elseif a.t=NORM then
    c ← (a.s=0) ? G : L
  elseif b.t=NORM then
    c ← (b.s=0) ? G : L
  elseif a.t=ZERO and b.t=ZERO then
    c ← E
  else
    assert FALSE // should have covered all the cases above
  endif
enddef

```

FIG. 25D-5

U.S. Patent

Apr. 20, 2004

Sheet 97 of 148

US 6,725,356 B2

2570

```

def c ← fmul(a,b) as
  if a.t=NORM and b.t=NORM then
    c.s ← a.s ^ b.s
    c.t ← NORM
    c.e ← a.e + b.e
    c.f ← a.f * b.f
  // priority is given to b operand for NaN propagation
  elseif (b.t=SNAN) or (b.t=QNAN) then
    c.s ← a.s ^ b.s
    c.t ← b.t
    c.e ← b.e
    c.f ← b.f
  elseif (a.t=SNAN) or (a.t=QNAN) then
    c.s ← a.s ^ b.s
    c.t ← a.t
    c.e ← a.e
    c.f ← a.f
  elseif a.t=ZERO and b.t=INFINITY then
    c ← DEFAULTSNAN // Invalid
  elseif a.t=INFINITY and b.t=ZERO then
    c ← DEFAULTSNAN // Invalid
  elseif a.t=ZERO or b.t=ZERO then
    c.s ← a.s ^ b.s
    c.t ← ZERO
  else
    assert FALSE // should have covered all the cases above
  endif
enddef

```

FIG. 25D-6

U.S. Patent

Apr. 20, 2004

Sheet 98 of 148

US 6,725,356 B2

2570

```

def c fdivr(a,b) as
  if a.t=NORM and b.t=NORM then
    c.s ← a.s ^ b.s
    c.t ← NORM
    c.e ← a.e - b.e + 256
    c.f ← (a.f 0 ) / b.f
  // priority is given to b operand for NaN propagation
  elseif (b.t=SNAN) or (b.t=QNAN) then
    c.s ← a.s ^ b.s
    c.t ← b.t
    c.e ← b.e
    c.f ← b.f
  elseif (a.t=SNAN) or (a.t=QNAN) then
    c.s ← a.s ^ b.s
    c.t ← a.t
    c.e ← a.e
    c.f ← a.f
  elseif a.t=ZERO and b.t=INFINITY then
    c ← DEFAULTSNAN // Invalid
  elseif a.t=INFINITY and b.t=INFINITY then
    c ← DEFAULTSNAN // Invalid
  elseif a.t=ZERO then
    c.s ← a.s ^ b.s
    c.t ← ZERO
  elseif a.t=INFINITY then
    c.s ← a.s ^ b.s
    c.t ← INFINITY
  else
    assert FALSE // should have covered al the cases above
  endif
enddef

def msb ← findmsb(a) as
  MAXF ← 218 // Largest possible f value after matrix multiply
  for j ← 0 to MAXF
    if aMAXF-1..j = (0MAXF-1-j || 1) then
      msb ← j
    endif
  endfor
enddef

```

FIG. 25D-7

U.S. Patent

Apr. 20, 2004

Sheet 99 of 148

US 6,725,356 B2

2570

```

Def ai ← PackF(prec,a,round) as
  case a.t of
    NORM:
      msb ← findmsb(a.f)
      m ← msb-1-fbits(prec) //1sb for normal
      rdn ← -ebias(prec)-a.e-1-fbits(prec) // 1sb if a denormal
      rb ← (m > rdn) ? m : rdn
      if rb < 0 then
        aifr ← a.fmsb-1..0 || 0-rb
        eadj ← 0
      else
        case round of
          C:
            s ← 0msb-rb || (~a.s)rb
          F:
            s ← 0msb-rb || (a.s)rb
          N, NONE:
            s ← 0msb-rb || ~a.frb || a.frb-1
          X:
            if a.frb-1..0 ≠ 0 then
              raise FloatingPointArithmetic // Inexact
            endif
            s ← 0
          Z:
            s ← 0
        endcase
      v ← (0 || a.fmsb..0) + (0 || s)
      if vmsb=1 then
        aifr ← vmsb-1..rb
        eadj ← 0
      else
        aifr ← 0fbits(prec)
        eadj ← 1
      endif
    endif
  aien ← a.e + msb - 1 + eadj + ebias(prec)
  if aien ≤ 0 then
    if round = NONE then
      ai ← a.s || 0ebits(prec) || aifr
    else
      raise FloatingPointArithmetic //Underflow
    endif
  endif

```

FIG. 25D-8

U.S. Patent

Apr. 20, 2004

Sheet 100 of 148

US 6,725,356 B2

2570

```

endif
elseif aien ≥ 1ebits(prec) then
  if round = NONE then
    //default: round-to-nearest overflow handling
    ai ← a.s || 1ebits(prec) || 0fbits(prec)
  else
    raise FloatingPointArithmetic // Overflow
  endif
endif
else
  ai ← a.s || aienebits(prec)-1..0 || aifr
endif

SNAN:
  if round ≠ NONE then
    raise FloatingPointArithmetic //Invalid
  endif
  if -a.e < fbits(prec) then
    ai ← a.s || 1ebits(prec) || a.f.a.e-1..0 || 0fbits(prec)+a.e
  else
    lsb ← a.f.a.e-1-fbits(prec)+1..0 ≠ 0
    ai ← a.s || 1ebits(prec) || a.f.a.e-1..a.e-1-fbits(prec)+2 || 1sb
  endif
endif

QNAN:
  if -a.e < fbits(prec) then
    ai ← a.s || 1ebits(prec) || a.f.a.e-1..0 || 0fbits(prec)+a.e
  else
    lsb ← a.f.a.e-1-fbits(prec)+1..0 ≠ 0
    ai ← a.s || 1ebits(prec) || a.f.a.e-1..a.e-1-fbits(prec)+2 || 1sb
  endif
endif

ZERO:
  ai ← a.s || 0ebits(prec) || 0fbits(prec)

INFINITY:
  ai ← a.s || 1ebits(prec) || 0fbits(prec)

endcase
defdef

```

FIG. 25D-9

U.S. Patent

Apr. 20, 2004

Sheet 101 of 148

US 6,725,356 B2

✓ 2570

```

Def ai ← fsinkr(prec, a, round) as
  case a.t of
    NORM:
      msb ← findmsb(a.f)
      rb ← -a.e
      if rb ≤ 0 then
        ai.fr ← a.fmsb..0 || 0-rb
        aims ← msb - rb
      else
        case round of
          C,C,D:
            s ← 0msb-rb || (~ai.s)rb
          F,F,D:
            s ← 0msb-rb || (ai.s)rb
          N, NONE:
            s ← 0msb-rb || ~ai.frb || ai.frbrb-1
          X:
            if ai.frb-1..0 ≠ 0 then
              raise FloatingPointArithmetic // Inexact
            endif
            s ← 0
          Z, Z,D:
            s ← 0
        endcase
      v ← (0 || a.fmsb..0) + (0 || s)
      if vmsb = 1 then
        aims ← msb + 1 - rb
      else
        aims ← msb - rb
      endif
      ai.fr ← vaims..rb
    endif
  if aims > prec then
    case round of
      C,D, F,D, NONE, Z,D:
        ai ← a.s || (~as)prec-1
      C,F,N,X,Z:
        raise FloatingPointArithmetic // Overflow
    endcase
  endif

```

FIG. 25D-10

U.S. Patent

Apr. 20, 2004

Sheet 102 of 148

US 6,725,356 B2

2570

```

elseif a.s = 0 then
    ai ← aifr
else
    ai ← -aifr
endif
ZERO:
    ai ← 0prec
SNAN, QNAN:
    case round of
        C.D, F.D, NONE, Z.D:
            ai ← 0prec
        C, F, N, X, Z:
            raise FloatingPoint Arithmetic // Invalid
    endcase
INFINITY:
    case round of
        C.D, F.D, NONE, Z.D:
            ai ← a.s || (-as)prec-1
        C, F, N, X, Z:
            raise FloatingPointArithmetic // Invalid
    endcase
endcase
enddef

def c ← frecrest(a) as
    b.s ← 0
    b.t ← NORM
    b.e ← 0
    b.f ← 1
    c ← fest(fdiv(b,a))
enddef

def c ← frsqrest(a) as
    b.s ← 0
    b.t ← NORM
    b.e ← 0
    b.f ← 1
    c ← fest(fsqr(fdiv(b,a)))
enddef

```

FIG. 25D-11

U.S. Patent

Apr. 20, 2004

Sheet 103 of 148

US 6,725,356 B2

2570

```

def c ← fest(a) as
  if (a.t=NORM) then
    msb ← findmsb(a.f)
    a.e ← a.e + msb - 13
    a.f ← a.fmsb..msb-12 || 1
  else
    c ← a
  endif
enddef

def ← fsqr(a) as
  if (a.t=NORM) and (a.s=0) then
    c.s ← 0
    c.t ← NORM
    if (a.e0 = 1) then
      c.e ← (a.e-127) / 2
      c.f ← sqr(a.f || 0127)
    else
      c.e ← (a.e-128) / 2
      c.f ← sqr(a.f || 0128)
    endif
  elseif (a.t=SNAN) or (a.t=QNAN) or a.t=ZERO or ((a.t=INFINITY) and
    (a.s=0)) then
    c ← a
  elseif ((a.t=NORM) or (a.t=INFINITY)) and (a.s=1) then
    c ← DEFAULTSNAN // Invalid
  else
    assert FALSE // should have covered all the cases above
  endif
enddef

```

FIG. 25D-12

U.S. Patent

Apr. 20, 2004

Sheet 104 of 148

US 6,725,356 B2

Operation codes

G.ADD.8	Group add bytes
G.ADD.16	Group add doublets
G.ADD.32	Group add quadlets
G.ADD.64	Group add octlets
G.ADD.128	Group add hexlet
G.ADD.L.8	Group add limit signed bytes
G.ADD.L.16	Group add limit signed doublets
G.ADD.L.32	Group add limit signed quadlets
G.ADD.L.64	Group add limit signed octlets
G.ADD.L.128	Group add limit signed hexlet
G.ADD.L.U.8	Group add limit unsigned bytes
G.ADD.L.U.16	Group add limit unsigned doublets
G.ADD.L.U.32	Group add limit unsigned quadlets
G.ADD.L.U.64	Group add limit unsigned octlets
G.ADD.L.U.128	Group add limit unsigned hexlet
G.ADD.8.O	Group add signed bytes check overflow
G.ADD.16.O	Group add signed doublets check overflow
G.ADD.32.O	Group add signed quadlets check overflow
G.ADD.64.O	Group add signed octlets check overflow
G.ADD.128.O	Group add signed hexlet check overflow
G.ADD.U.8.O	Group add unsigned bytes check overflow
G.ADD.U.16.O	Group add unsigned doublets check overflow
G.ADD.U.32.O	Group add unsigned quadlets check overflow
G.ADD.U.64.O	Group add unsigned octlets check overflow
G.ADD.U.128.O	Group add unsigned hexlet check overflow

FIG. 26A

U.S. Patent

Apr. 20, 2004

Sheet 105 of 148

US 6,725,356 B2

Format

G.op.size rd=rc,rb

rd=gopsize(rc,rb)

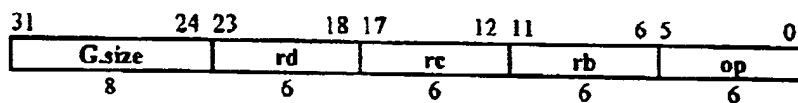


FIG. 26B

U.S. Patent

Apr. 20, 2004

Sheet 106 of 148

US 6,725,356 B2

Definition

```

def Group(op,size,rd,rc,rb)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case op of
    G.ADD:
      for i ← 0 to 128-size by size
        ai+size-1..i ← ci+size-1..i + bi+size-1..i
      endfor
    G.ADD.L:
      for i ← 0 to 128-size by size
        t ← (ci+size-1 || ci+size-1..i) + (bi+size-1 || bi+size-1..i)
        ai+size-1..i ← (tsize ≠ tsize-1) ? (tsize || tsize-1) : tsize-1..0
      endfor
    G.ADD.L.U:
      for i ← 0 to 128-size by size
        t ← (01 || ci+size-1..i) + (01 || bi+size-1..i)
        ai+size-1..i ← (tsize ≠ 0) ? (1size) : tsize-1..0
      endfor
    G.ADD.O:
      for i ← 0 to 128-size by size
        t ← (ci+size-1 || ci+size-1..i) + (bi+size-1 || bi+size-1..i)
        if tsize ≠ tsize-1 then
          raise FixedPointArithmetic
        endif
        ai+size-1..i ← tsize-1..0
      endfor
    G.ADD.U.O:
      for i ← 0 to 128-size by size
        t ← (01 || ci+size-1..i) + (01 || bi+size-1..i)
        if tsize ≠ 0 then
          raise FixedPointArithmetic
        endif
        ai+size-1..i ← tsize-1..0
      endfor
  endcase
  RegWrite(rd, 128, a)
enddef

```

FIG. 26C

U.S. Patent

Apr. 20, 2004

Sheet 107 of 148

US 6,725,356 B2

Operation codes

G.SET.AND.E.8	Group set and equal zero bytes
G.SET.AND.E.16	Group set and equal zero doublets
G.SET.AND.E.32	Group set and equal zero quadlets
G.SET.AND.E.64	Group set and equal zero octlets
G.SET.AND.E.128	Group set and equal zero hexlet
G.SET.AND.NE.8	Group set and not equal zero bytes
G.SET.AND.NE.16	Group set and not equal zero doublets
G.SET.AND.NE.32	Group set and not equal zero quadlets
G.SET.AND.NE.64	Group set and not equal zero octlets
G.SET.AND.NE.128	Group set and not equal zero hexlet
G.SET.E.8	Group set equal bytes
G.SET.E.16	Group set equal doublets
G.SET.E.32	Group set equal quadlets
G.SET.E.64	Group set equal octlets
G.SET.E.128	Group set equal hexlet
G.SET.GE.8	Group set greater equal signed bytes
G.SET.GE.16	Group set greater equal signed doublets
G.SET.GE.32	Group set greater equal signed quadlets
G.SET.GE.64	Group set greater equal signed octlets
G.SET.GE.128	Group set greater equal signed hexlet
G.SET.GE.U.8	Group set greater equal unsigned bytes
G.SET.GE.U.16	Group set greater equal unsigned doublets
G.SET.GE.U.32	Group set greater equal unsigned quadlets
G.SET.GE.U.64	Group set greater equal unsigned octlets
G.SET.GE.U.128	Group set greater equal unsigned hexlet
G.SET.L.8	Group set signed less bytes
G.SET.L.16	Group set signed less doublets
G.SET.L.32	Group set signed less quadlets
G.SET.L.64	Group set signed less octlets
G.SET.L.128	Group set signed less hexlet
G.SET.L.U.8	Group set less unsigned bytes
G.SET.L.U.16	Group set less unsigned doublets
G.SET.L.U.32	Group set less unsigned quadlets
G.SET.L.U.64	Group set less unsigned octlets
G.SET.L.U.128	Group set less unsigned hexlet
G.SET.NE.8	Group set not equal bytes
G.SET.NE.16	Group set not equal doublets
G.SET.NE.32	Group set not equal quadlets
G.SET.NE.64	Group set not equal octlets
G.SET.NE.128	Group set not equal hexlet
G.SUB.8	Group subtract bytes
G.SUB.8.0	Group subtract signed bytes check overflow

FIG. 27A-1

U.S. Patent

Apr. 20, 2004

Sheet 108 of 148

US 6,725,356 B2

G.SUB.16	Group subtract doublets
G.SUB.16.O	Group subtract signed doublets check overflow
G.SUB.32	Group subtract quadlets
G.SUB.32.O	Group subtract signed quadlets check overflow
G.SUB.64	Group subtract octlets
G.SUB.64.O	Group subtract signed octlets check overflow
G.SUB.128	Group subtract hexlet
G.SUB.128.O	Group subtract signed hexlet check overflow
G.SUB.L.8	Group subtract limit signed bytes
G.SUB.L.16	Group subtract limit signed doublets
G.SUB.L.32	Group subtract limit signed quadlets
G.SUB.L.64	Group subtract limit signed octlets
G.SUB.L.128	Group subtract limit signed hexlet
G.SUB.L.U.8	Group subtract limit unsigned bytes
G.SUB.L.U.16	Group subtract limit unsigned doublets
G.SUB.L.U.32	Group subtract limit unsigned quadlets
G.SUB.L.U.64	Group subtract limit unsigned octlets
G.SUB.L.U.128	Group subtract limit unsigned hexlet
G.SUB.U.8.O	Group subtract unsigned bytes check overflow
G.SUB.U.16.O	Group subtract unsigned doublets check overflow
G.SUB.U.32.O	Group subtract unsigned quadlets check overflow
G.SUB.U.64.O	Group subtract unsigned octlets check overflow
G.SUB.U.128.O	Group subtract unsigned hexlet check overflow

FIG. 27A-2

U.S. Patent

Apr. 20, 2004

Sheet 109 of 148

US 6,725,356 B2

Format

G.op.size rd=rb,rc

rd=gopsize(rb,rc)

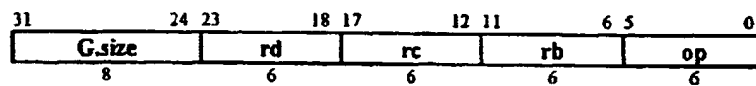


FIG. 27B

U.S. Patent

Apr. 20, 2004

Sheet 110 of 148

US 6,725,356 B2

Definition

```

def GroupReversed(op,size,rd,rc,rb)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case op of
    G.SUB:
      for i ← 0 to 128-size by size
        ai+size-1..i ← bi+size-1..i - ci+size-1..i
      endfor
    G.SUB.L:
      for i ← 0 to 128-size by size
        t ← (bi+size-1 || bi+size-1..i) - (ci+size-1 || ci+size-1..i)
        ai+size-1..i ← (tsize ≠ tsize-1) ? (tsize || (tsize-1)) : tsize-1..0
      endfor
    G.SUB.LU:
      for i ← 0 to 128-size by size
        t ← (01 || bi+size-1..i) - (01 || ci+size-1..i)
        ai+size-1..i ← (tsize ≠ 0) ? 0size : tsize-1..0
      endfor
    G.SUB.O:
      for i ← 0 to 128-size by size
        t ← (bi+size-1 || bi+size-1..i) - (ci+size-1 || ci+size-1..i)
        if (tsize ≠ tsize-1) then
          raise FixedPointArithmetic
        endif
        ai+size-1..i ← tsize-1..0
      endfor
    G.SUB.U.O:
      for i ← 0 to 128-size by size
        t ← (01 || bi+size-1..i) - (01 || ci+size-1..i)
        if (tsize ≠ 0) then
          raise FixedPointArithmetic
        endif
        ai+size-1..i ← tsize-1..0
      endfor
    G.SET.E:
      for i ← 0 to 128-size by size
        ai+size-1..i ← (bi+size-1..i = ci+size-1..i)size
      endfor
    G.SET.NE:
      for i ← 0 to 128-size by size
        ai+size-1..i ← (bi+size-1..i ≠ ci+size-1..i)size
      endfor
    G.SET.AND.E:
      for i ← 0 to 128-size by size
        ai+size-1..i ← ((bi+size-1..i and ci+size-1..i) = 0)size
      endfor
  endcase
enddef

```

FIG. 27C-1

U.S. Patent

Apr. 20, 2004

Sheet 111 of 148

US 6,725,356 B2

```

G.SET.AND.NE:
  for i ← 0 to 128-size by size
    ai+size-1..i ← ((bi+size-1..i and ci+size-1..i) ≠ 0)size
  endfor
G.SET.L:
  for i ← 0 to 128-size by size
    ai+size-1..i ← ((rc = rb) ? (bi+size-1..i < 0) : (bi+size-1..i < ci+size-1..i))size
  endfor
G.SET.GE:
  for i ← 0 to 128-size by size
    ai+size-1..i ← ((rc = rb) ? (bi+size-1..i ≥ 0) : (bi+size-1..i ≥ ci+size-1..i))size
  endfor
G.SET.L.U:
  for i ← 0 to 128-size by size
    ai+size-1..i ← ((rc = rb) ? (bi+size-1..i > 0) :
      ((0 || bi+size-1..i) < (0 || ci+size-1..i)))size
  endfor
G.SET.GE.U:
  for i ← 0 to 128-size by size
    ai+size-1..i ← ((rc = rb) ? (bi+size-1..i ≤ 0) :
      ((0 || bi+size-1..i) ≥ (0 || ci+size-1..i)))size
  endfor
endcase
RegWrite(rd, 128, a)
enddef

```

FIG. 27C-2

U.S. Patent

Apr. 20, 2004

Sheet 112 of 148

US 6,725,356 B2

Operation codes

E.CON.8	Ensemble convolve signed bytes
E.CON.16	Ensemble convolve signed doublets
E.CON.32	Ensemble convolve signed quadlets
E.CON.64	Ensemble convolve signed octlets
E.CON.C.8	Ensemble convolve complex bytes
E.CON.C.16	Ensemble convolve complex doublets
E.CON.C.32	Ensemble convolve complex quadlets
E.CON.M.8	Ensemble convolve mixed-signed bytes
E.CON.M.16	Ensemble convolve mixed-signed doublets
E.CON.M.32	Ensemble convolve mixed-signed quadlets
E.CON.M.64	Ensemble convolve mixed-signed octlets
E.CON.U.8	Ensemble convolve unsigned bytes
E.CON.U.16	Ensemble convolve unsigned doublets
E.CON.U.32	Ensemble convolve unsigned quadlets
E.CON.U.64	Ensemble convolve unsigned octlets
E.DIV.64	Ensemble divide signed octlets
E.DIV.U.64	Ensemble divide unsigned octlets
E.MUL.8	Ensemble multiply signed bytes
E.MUL.16	Ensemble multiply signed doublets
E.MUL.32	Ensemble multiply signed quadlets
E.MUL.64	Ensemble multiply signed octlets
E.MUL.SUM.8	Ensemble multiply sum signed bytes
E.MUL.SUM.16	Ensemble multiply sum signed doublets
E.MUL.SUM.32	Ensemble multiply sum signed quadlets
E.MUL.SUM.64	Ensemble multiply sum signed octlets
E.MUL.C.8	Ensemble complex multiply bytes
E.MUL.C.16	Ensemble complex multiply doublets
E.MUL.C.32	Ensemble complex multiply quadlets
E.MUL.M.8	Ensemble multiply mixed-signed bytes
E.MUL.M.16	Ensemble multiply mixed-signed doublets
E.MUL.M.32	Ensemble multiply mixed-signed quadlets
E.MUL.M.64	Ensemble multiply mixed-signed octlets
E.MUL.P.8	Ensemble multiply polynomial bytes
E.MUL.P.16	Ensemble multiply polynomial doublets
E.MUL.P.32	Ensemble multiply polynomial quadlets
E.MUL.P.64	Ensemble multiply polynomial octlets
E.MUL.SUM.C.8	Ensemble multiply sum complex bytes
E.MUL.SUM.C.16	Ensemble multiply sum complex doublets
E.MUL.SUM.C.32	Ensemble multiply sum complex quadlets
E.MUL.SUM.M.8	Ensemble multiply sum mixed-signed bytes
E.MUL.SUM.M.16	Ensemble multiply sum mixed-signed doublets
E.MUL.SUM.M.32	Ensemble multiply sum mixed-signed quadlets
E.MUL.SUM.M.64	Ensemble multiply sum mixed-signed octlets

FIG. 28A-1

U.S. Patent

Apr. 20, 2004

Sheet 113 of 148

US 6,725,356 B2

E.MUL.SUM.U.8	Ensemble multiply sum unsigned bytes
E.MUL.SUM.U.16	Ensemble multiply sum unsigned doublets
E.MUL.SUM.U.32	Ensemble multiply sum unsigned quadlets
E.MUL.SUM.U.64	Ensemble multiply sum unsigned octlets
E.MUL.U.8	Ensemble multiply unsigned bytes
E.MUL.U.16	Ensemble multiply unsigned doublets
E.MUL.U.32	Ensemble multiply unsigned quadlets
E.MUL.U.64	Ensemble multiply unsigned octlets

FIG. 28A-2

U.S. Patent

Apr. 20, 2004

Sheet 114 of 148

US 6,725,356 B2

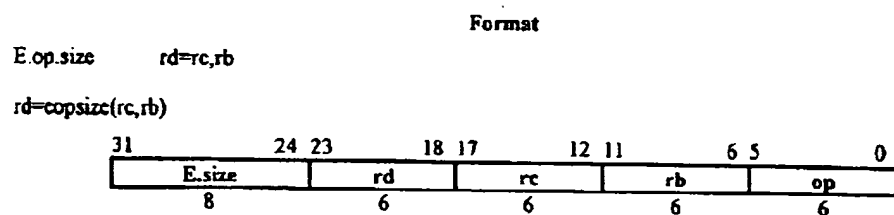


FIG. 28B

U.S. Patent

Apr. 20, 2004

Sheet 115 of 148

US 6,725,356 B2

Definition

```

def mul(size,h,vs,v,i,ws,w,j) as
    mul ← ((vs&vsize-1+i)h-size || vsize-1+i..i) * ((ws&wsize-1+j)h-size || wsize-1+j..j)
enddef

def c ← PolyMultiply(size,a,b) as
    p[0] ← 02*size
    for k ← 0 to size-1
        p[k+1] ← p[k] ^ ak ? (0size-k || b || 0k) : 02*size
    endfor
    c ← p[size]
enddef

def Ensemble(op,size,rd,rc,rb)
    c ← RegRead(rc, 128)
    b ← RegRead(rb, 128)
    case op of
        E.MUL:, E.MUL.C:, EMUL.SUM, E.MUL.SUM.C, E.CON, E.CON.C, E.DIV:
            cs ← bs ← 1
        E.MUL.M:, EMUL.SUM.M, E.CON.M:
            cs ← 0
            bs ← 1
        E.MUL.U:, EMUL.SUM.U, E.CON.U, E.DIV.U, E.MUL.P:
            cs ← bs ← 0
    endcase
    case op of
        E.MUL, E.MUL.U, E.MUL.M:
            for i ← 0 to 64-size by size
                d2*(i+size)-1..2*i ← mul(size,2*size,cs,c,i,bs,b,i)
            endfor
        E.MUL.P:
            for i ← 0 to 64-size by size
                d2*(i+size)-1..2*i ← PolyMultiply(size,csize-1+i..i,bsize-1+i..i)
            endfor
        E.MUL.C:
            for i ← 0 to 64-size by size
                if (i and size) = 0 then
                    p ← mul(size,2*size,1,c,i,1,b,i) - mul(size,2*size,1,c,i+size,1,b,i+size)
                else
                    p ← mul(size,2*size,1,c,i,1,b,i+size) + mul(size,2*size,1,c,i,1,b,i+size)
                endif
                d2*(i+size)-1..2*i ← p
            endfor
        EMUL.SUM, E.MUL.SUM.U, E.MUL.SUM.M:
            p[0] ← 0128
            for i ← 0 to 128-size by size
                p[i+size] ← p[i] + mul(size,128,cs,c,i,bs,b,i)
            endfor
    endcase
enddef

```

FIG. 28C-1

U.S. Patent

Apr. 20, 2004

Sheet 116 of 148

US 6,725,356 B2

```

a ← p[128]
E.MUL.SUM.C:
  p[0] ← 064
  p[size] ← 064
  for i ← 0 to 128-size by size
    if (i and size) = 0 then
      p[i+2*size] ← p[i] + mul(size,64,1,c,i,1,b,i)
                      - mul(size,64,1,c,i+size,1,b,i+size)
    else
      p[i+2*size] ← p[i] + mul(size,64,1,c,i,1,b,i+size)
                      + mul(size,64,1,c,i+size,1,b,i)
    endif
  endfor
  a ← p[128+size] || p[128]
E.CON, E.CON.U, E.CON.M:
  p[0] ← 0128
  for j ← 0 to 64-size by size
    for i ← 0 to 64-size by size
      p[j+size]2*(i+size)-1..2*i ← p[j]2*(i+size)-1..2*i +
        mul(size,2*size,cs,c,i+64-j,bs,b,j)
    endfor
  endfor
  a ← p[64]
E.CON.C:
  p[0] ← 0128
  for j ← 0 to 64-size by size
    for i ← 0 to 64-size by size
      if ((~i) and j and size) = 0 then
        p[j+size]2*(i+size)-1..2*i ← p[j]2*(i+size)-1..2*i +
          mul(size,2*size,1,c,i+64-j,1,b,j)
      else
        p[j+size]2*(i+size)-1..2*i ← p[j]2*(i+size)-1..2*i -
          mul(size,2*size,1,c,i+64-j+2*size,1,b,j)
      endif
    endfor
  endfor
  a ← p[64]
E.DIV:
  if (b = 0) or ((c = (1||063)) and (b = 164)) then
    a ← undefined
  
```

FIG. 28C-2

U.S. Patent

Apr. 20, 2004

Sheet 117 of 148

US 6,725,356 B2

```
else
    q ← c / b
    r ← c - q*b
    a ← r63..0 || q63..0
endif
E.DIV.U:
if b = 0 then
    a ← undefined
else
    q ← (0 || c) / (0 || b)
    r ← c - (0 || q)*(0 || b)
    a ← r63..0 || q63..0
endif
endcase
RegWrite(rd, 128, a)
enddef
```

FIG. 28C-3

U.S. Patent

Apr. 20, 2004

Sheet 118 of 148

US 6,725,356 B2

Floating-point function Definitions

```

def eb ← ebits(prec) as
  case pref of
    16:
      eb ← 5
    32:
      eb ← 8
    64:
      eb ← 11
    128:
      eb ← 15
  endcase
enddef

def eb ← ebias(prec) as
  eb ← 0 || 1ebits(prec)-1
enddef

def fb ← fbits(prec) as
  fb ← prec - 1 - eb
enddef

def a ← F(prec, ai) as
  a.s ← aiprec-1
  ae ← aiprec-2..fbits(prec)
  af ← aifbits(prec)-1..0
  if ae = 1ebits(prec) then
    if af = 0 then
      a.t ← INFINITY
    elseif afbits(prec)-1 then
      a.t ← SNaN
      a.e ← -fbits(prec)
      a.f ← 1 || afbits(prec)-2..0
    else
      a.t ← QNaN
      a.e ← -fbits(prec)
      a.f ← af
    endif
  endif
enddef

```

FIG. 29-1

U.S. Patent

Apr. 20, 2004

Sheet 119 of 148

US 6,725,356 B2

```

elseif ae = 0 then
  if af = 0 then
    a.t ← ZERO
  else
    a.t ← NORM
    a.e ← 1-ebias(prec)-fbits(prec)
    a.f ← 0 || af
  endif
endif
else
  a.t ← NORM
  a.e ← ae-ebias(prec)-fbits(prec)
  a.f ← 1 || af
endif
enddef

def a ← DEFAULTQNaN as
  a.s ← 0
  a.t ← QNaN
  a.e ← -1
  a.f ← 1
enddef

def a ← DEFAULTSNAN as
  a.s ← 0
  a.t ← SNAN
  a.e ← -1
  a.f ← 1
enddef

def fadd(a,b) as faddr(a,b,N) enddef

def c ← faddr(a,b,round) as
  if a.t=NORM and b.t=NORM then
    // d,e are a,b with exponent aligned and fraction adjusted
    if a.e > b.e then
      d ← a
      e.t ← b.t
      e.s ← b.s
      e.e ← a.e
      e.f ← b.f || 0a.e-b.e
    else if a.e < b.e then
      d.t ← a.t
      d.s ← a.s
      d.e ← b.e
      d.f ← a.f || 0b.e-a.e
      e ← b
    end
  end
end

```

FIG. 29-2

U.S. Patent

Apr. 20, 2004

Sheet 120 of 148

US 6,725,356 B2

```

endif
c.t ← d.t
c.e ← d.e
if d.s = e.s then
    c.s ← d.s
    c.f ← d.f + e.f
elseif d.f > e.f then
    c.s ← d.s
    c.f ← d.f - e.f
elseif d.f < e.f then
    c.s ← e.s
    c.f ← e.f - d.f
else
    c.s ← r=F
    c.t ← ZERO
endif
// priority is given to b operand for NaN propagation
elseif (b.t=SNAN) or (b.t=QNAN) then
    c ← b
elseif (a.t=SNAN) or (a.t=QNAN) then
    c ← a
elseif a.t=ZERO and b.t=ZERO then
    c.t ← ZERO
    c.s ← (a.s and b.s) or (round=F and (a.s or b.s))
// NULL values are like zero, but do not combine with ZERO to alter sign
elseif a.t=ZERO or a.t=NULL then
    c ← b
elseif b.t=ZERO or b.t=NULL then
    c ← a
elseif a.t=INFINITY and b.t=INFINITY then
    if a.s ≠ b.s then
        c ← DEFAULTSNAN // Invalid
    else
        c ← a
    endif
elseif a.t=INFINITY then
    c ← a
elseif b.t=INFINITY then
    c ← b
else
    assert FALSE // should have covered all the cases above
endif
enddef

def b ← fneg(a) as
    b.s ← ~a.s
    b.t ← a.t
    b.e ← a.e
    b.f ← a.f
enddef

```

FIG. 29-3

U.S. Patent

Apr. 20, 2004

Sheet 121 of 148

US 6,725,356 B2

```

def fsubr(a,b,round) as faddr(a,fneg(b),round) enddef

def frsub(a,b) as frsubr(a,b,N) enddef

def frsubr(a,b,round) as faddr(fneg(a),b,round) enddef

def c ← fcom(a,b) as
  if (a.t=SNAN) or (a.t=QNAN) or (b.t=SNAN) or (b.t=QNAN) then
    c ← U
  elseif a.t=INFINITY and b.t=INFINITY then
    if a.s ≠ b.s then
      c ← (a.s=0) ? G: L
    else
      c ← E
    endif
  elseif a.t=INFINITY then
    c ← (a.s=0) ? G: L
  elseif b.t=INFINITY then
    c ← (b.s=0) ? G: L
  elseif a.t=NORM and b.t=NORM then
    if a.s ≠ b.s then
      c ← (a.s=0) ? G: L
    else
      if a.e > b.e then
        af ← a.f
        bf ← b.f || 0a.e-b.e
      else
        af ← a.f || 0b.e-a.e
        bf ← b.f
      endif
      if af = bf then
        c ← E
      else
        c ← ((a.s=0) ^ (af > bf)) ? G : L
      endif
    endif
  elseif a.t=NORM then
    c ← (a.s=0) ? G: L
  elseif b.t=NORM then
    c ← (b.s=0) ? G: L
  elseif a.t=ZERO and b.t=ZERO then
    c ← E
  else
    assert FALSE // should have covered all the cases above
  endif
enddef

```

FIG. 29-4

U.S. Patent

Apr. 20, 2004

Sheet 122 of 148

US 6,725,356 B2

```

def c ← fmul(a,b) as
  if a.t=NORM and b.t=NORM then
    c.s ← a.s ^ b.s
    c.t ← NORM
    c.e ← a.e + b.e
    c.f ← a.f * b.f
    // priority is given to b operand for NaN propagation
    elseif (b.t=SNAN) or (b.t=QNAN) then
      c.s ← a.s ^ b.s
      c.t ← b.t
      c.e ← b.e
      c.f ← b.f
    elseif (a.t=SNAN) or (a.t=QNAN) then
      c.s ← a.s ^ b.s
      c.t ← a.t
      c.e ← a.e
      c.f ← a.f
    elseif a.t=ZERO and b.t=INFINITY then
      c ← DEFAULTSNAN // Invalid
    elseif a.t=INFINITY and b.t=ZERO then
      c ← DEFAULTSNAN // Invalid
    elseif a.t=ZERO or b.t=ZERO then
      c.s ← a.s ^ b.s
      c.t ← ZERO
    else
      assert FALSE // should have covered all the cases above
    endif
  endif
enddef

def c ← fdivr(a,b) as
  if a.t=NORM and b.t=NORM then
    c.s ← a.s ^ b.s
    c.t ← NORM
    c.e ← a.e - b.e + 256
    c.f ← (a.f || 0256) / b.f
    // priority is given to b operand for NaN propagation
    elseif (b.t=SNAN) or (b.t=QNAN) then
      c.s ← a.s ^ b.s
      c.t ← b.t
      c.e ← b.e
      c.f ← b.f
    elseif (a.t=SNAN) or (a.t=QNAN) then
      c.s ← a.s ^ b.s
      c.t ← a.t
      c.e ← a.e
      c.f ← a.f

```

FIG. 29-5

U.S. Patent

Apr. 20, 2004

Sheet 123 of 148

US 6,725,356 B2

```

elseif a.t=ZERO and b.t=ZERO then
  c ← DEFAULTSNAN // Invalid
elseif a.t=INFINITY and b.t=INFINITY then
  c ← DEFAULTSNAN // Invalid
elseif a.t=ZERO then
  c.s ← a.s ^ b.s
  c.t ← ZERO
elseif a.t=INFINITY then
  c.s ← a.s ^ b.s
  c.t ← INFINITY
else
  assert FALSE // should have covered all the cases above
endif
enddef

def msb ← findmsb(a) as
  MAXF ← 218 // Largest possible f value after matrix multiply
  for j ← 0 to MAXF
    if a[MAXF-1..j] = (0MAXF-1-j || 1) then
      msb ← j
    endif
  endfor
enddef

def ai ← PackF(prec,a,round) as
  case a.t of
    NORM:
      msb ← findmsb(a.f)
      rn ← msb-1-fbits(prec) // lsb for normal
      rdn ← -ebias(prec)-a.e-1-fbits(prec) // lsb if a denormal
      rb ← (rn > rdn) ? rn : rdn

```

FIG. 29-6

U.S. Patent

Apr. 20, 2004

Sheet 124 of 148

US 6,725,356 B2

```

if rb ≤ 0 then
    aifr ← a.fmsb-1..0 || 0-rb
    eadj ← 0
else
    case round of
        C:
            s ← 0msb-rb || (~a.s)rb
        F:
            s ← 0msb-rb || (a.s)rb
        N, NONE:
            s ← 0msb-rb || ~a.frb || a.frb-1
        X:
            if a.frb-1..0 ≠ 0 then
                raise FloatingPointArithmetic // Inexact
            endif
            s ← 0
        Z:
            s ← 0
    endcase
    v ← (0||a.fmsb..0) + (0||s)
    if vmsb = 1 then
        aifr ← vmsb-1..rb
        eadj ← 0
    else
        aifr ← 0fbits(prec)
        eadj ← 1
    endif
endif
aien ← a.e + msb - 1 + eadj + ebias(prec)
if aien ≤ 0 then
    if round = NONE then
        ai ← a.s || 0ebits(prec) || aifr
    else
        raise FloatingPointArithmetic // Underflow
    endif
elseif aien ≥ 1ebits(prec) then
    if round = NONE then
        //default: round-to-nearest overflow handling
        ai ← a.s || 1ebits(prec) || 0fbits(prec)
    else
        raise FloatingPointArithmetic // Underflow
    endif
else
    ai ← a.s || aienebits(prec)-1..0 || aifr
endif

```

FIG. 29-7

U.S. Patent

Apr. 20, 2004

Sheet 125 of 148

US 6,725,356 B2

```

SNAN:
  if round ≠ NONE then
    raise FloatingPointArithmetic //Invalid
  endif
  if -a.e < fbits(prec) then
    ai ← a.s || 1ebits(prec) || a.f-a.e-1..0 || 0fbits(prec)+a.e
  else
    lsb ← a.f-a.e-1-fbits(prec)+1..0 ≠ 0
    ai ← a.s || 1ebits(prec) || a.f-a.e-1..a.e-1-fbits(prec)+2 || lsb
  endif
QNAN:
  if -a.e < fbits(prec) then
    ai ← a.s || 1ebits(prec) || a.f-a.e-1..0 || 0fbits(prec)+a.e
  else
    lsb ← a.f-a.e-1-fbits(prec)+1..0 ≠ 0
    ai ← a.s || 1ebits(prec) || a.f-a.e-1..a.e-1-fbits(prec)+2 || lsb
  endif
ZERO:
  ai ← a.s || 0fbits(prec) || 0fbits(prec)
INFINITY:
  ai ← a.s || 1ebits(prec) || 0fbits(prec)
endcase
defdef
def ai ← fsinkr(prec, a, round) as
  case a.t of
    NORM:
      msb ← findmsb(a.f)
      rb ← -a.e
      if rb ≤ 0 then
        aifr ← a.fmsb..0 || 0-rb
        aims ← msb - rb
      else
        case round of
          C, C.D:
            s ← 0msb-rb || (~ai.s)rb
          F, F.D:
            s ← 0msb-rb || (ai.s)rb
          N, NONE:
            s ← 0msb-rb || ~ai.frb || ai.frb-1
          X:
            if ai.frb-1..0 ≠ 0 then
              raise FloatingPointArithmetic // Inexact
            endif
            s ← 0
          Z, Z.D:
            s ← 0
        endcase
      endcase
    endcase
  enddef

```

FIG. 29-8

U.S. Patent

Apr. 20, 2004

Sheet 126 of 148

US 6,725,356 B2

```

        endcase
        v ← (0||a.f.msb..0) + (0||s)
        if v.msb = 1 then
            aims ← msb + 1 - rb
        else
            aims ← msb - rb
        endif
        aifr ← v.aims..rb
    endif
    if aims > prec then
        case round of
            C.D, F.D, NONE, Z.D:
                ai ← a.s || (~as)prec-1

            C, F, N, X, Z:
                raise FloatingPointArithmetic // Overflow
        endcase
    elseif a.s = 0 then
        ai ← aifr
    else
        ai ← -aifr
    endif
ZERO:
    ai ← 0prec
SNAN, QNAN:
    case round of
        C.D, F.D, NONE, Z.D:
            ai ← 0prec
        C, F, N, X, Z:
            raise FloatingPointArithmetic // Invalid
    endcase
INFINITY:
    case round of
        C.D, F.D, NONE, Z.D:
            ai ← a.s || (~as)prec-1
        C, F, N, X, Z:
            raise FloatingPointArithmetic // Invalid
    endcase
endcase
enddef

def c ← frecrest(a) as
    b.s ← 0
    b.t ← NORM
    b.c ← 0
    b.f ← 1
    c ← fest(fdiv(b,a))
enddef

```

FIG. 29-9

U.S. Patent

Apr. 20, 2004

Sheet 127 of 148

US 6,725,356 B2

```

def c ← frsqrest(a) as
  b.s ← 0
  b.t ← NORM
  b.e ← 0
  b.f ← 1
  c ← fest(fsq(fdiv(b,a)))
enddef

def c ← fest(a) as
  if (a.t=NORM) then
    msb ← findmsb(a.f)
    a.e ← a.e + msb - 13
    a.f ← a.f.msb.msb-12 || 1
  else
    c ← a
  endif
enddef

def c ← fsqr(a) as
  if (a.t=NORM) and (a.s=0) then
    c.s ← 0
    c.t ← NORM
    if (a.c0 = 1) then
      c.e ← (a.e-127) / 2
      c.f ← sqr(a.f || 0127)
    else
      c.e ← (a.e-128) / 2
      c.f ← sqr(a.f || 0128)
    endif
  elseif (a.t=SNAN) or (a.t=QNAN) or a.t=ZERO or ((a.t=INFINITY) and (a.s=0)) then
    c ← a
  elseif ((a.t=NORM) or (a.t=INFINITY)) and (a.s=1) then
    c ← DEFAULTSNAN // Invalid
  else
    assert FALSE // should have covered all the cases above
  endif
enddef

```

FIG. 29-10

U.S. Patent

Apr. 20, 2004

Sheet 128 of 148

US 6,725,356 B2

Operation codes

E.ADD.F.16	Ensemble add floating-point half
E.ADD.F.16.C	Ensemble add floating-point half ceiling
E.ADD.F.16.F	Ensemble add floating-point half floor
E.ADD.F.16.N	Ensemble add floating-point half nearest
E.ADD.F.16.X	Ensemble add floating-point half exact
E.ADD.F.16.Z	Ensemble add floating-point half zero
E.ADD.F.32	Ensemble add floating-point single
E.ADD.F.32.C	Ensemble add floating-point single ceiling
E.ADD.F.32.F	Ensemble add floating-point single floor
E.ADD.F.32.N	Ensemble add floating-point single nearest
E.ADD.F.32.X	Ensemble add floating-point single exact
E.ADD.F.32.Z	Ensemble add floating-point single zero
E.ADD.F.64	Ensemble add floating-point double
E.ADD.F.64.C	Ensemble add floating-point double ceiling
E.ADD.F.64.F	Ensemble add floating-point double floor
E.ADD.F.64.N	Ensemble add floating-point double nearest
E.ADD.F.64.X	Ensemble add floating-point double exact
E.ADD.F.64.Z	Ensemble add floating-point double zero
E.ADD.F.128	Ensemble add floating-point quad
E.ADD.F.128.C	Ensemble add floating-point quad ceiling
E.ADD.F.128.F	Ensemble add floating-point quad floor
E.ADD.F.128.N	Ensemble add floating-point quad nearest
E.ADD.F.128.X	Ensemble add floating-point quad exact
E.ADD.F.128.Z	Ensemble add floating-point quad zero
E.DIV.F.16	Ensemble divide floating-point half
E.DIV.F.16.C	Ensemble divide floating-point half ceiling
E.DIV.F.16.F	Ensemble divide floating-point half floor
E.DIV.F.16.N	Ensemble divide floating-point half nearest
E.DIV.F.16.X	Ensemble divide floating-point half exact
E.DIV.F.16.Z	Ensemble divide floating-point half zero
E.DIV.F.32	Ensemble divide floating-point single
E.DIV.F.32.C	Ensemble divide floating-point single ceiling
E.DIV.F.32.F	Ensemble divide floating-point single floor
E.DIV.F.32.N	Ensemble divide floating-point single nearest
E.DIV.F.32.X	Ensemble divide floating-point single exact
E.DIV.F.32.Z	Ensemble divide floating-point single zero
E.DIV.F.64	Ensemble divide floating-point double

FIG. 30A-1

U.S. Patent

Apr. 20, 2004

Sheet 129 of 148

US 6,725,356 B2

E.DIV.F.64.C	Ensemble divide floating-point double ceiling
E.DIV.F.64.F	Ensemble divide floating-point double floor
E.DIV.F.64.N	Ensemble divide floating-point double nearest
E.DIV.F.64.X	Ensemble divide floating-point double exact
E.DIV.F.64.Z	Ensemble divide floating-point double zero
E.DIV.F.128	Ensemble divide floating-point quad
E.DIV.F.128.C	Ensemble divide floating-point quad ceiling
E.DIV.F.128.F	Ensemble divide floating-point quad floor
E.DIV.F.128.N	Ensemble divide floating-point quad nearest
E.DIV.F.128.X	Ensemble divide floating-point quad exact
E.DIV.F.128.Z	Ensemble divide floating-point quad zero
E.MUL.C.F.16	Ensemble multiply complex floating-point half
E.MUL.C.F.32	Ensemble multiply complex floating-point single
E.MUL.C.F.64	Ensemble multiply complex floating-point double
E.MUL.F.16	Ensemble multiply floating-point half
E.MUL.F.16.C	Ensemble multiply floating-point half ceiling
E.MUL.F.16.F	Ensemble multiply floating-point half floor
E.MUL.F.16.N	Ensemble multiply floating-point half nearest
E.MUL.F.16.X	Ensemble multiply floating-point half exact
E.MUL.F.16.Z	Ensemble multiply floating-point half zero
E.MUL.F.32	Ensemble multiply floating-point single
E.MUL.F.32.C	Ensemble multiply floating-point single ceiling
E.MUL.F.32.F	Ensemble multiply floating-point single floor
E.MUL.F.32.N	Ensemble multiply floating-point single nearest
E.MUL.F.32.X	Ensemble multiply floating-point single exact
E.MUL.F.32.Z	Ensemble multiply floating-point single zero
E.MUL.F.64	Ensemble multiply floating-point double
E.MUL.F.64.C	Ensemble multiply floating-point double ceiling
E.MUL.F.64.F	Ensemble multiply floating-point double floor
E.MUL.F.64.N	Ensemble multiply floating-point double nearest
E.MUL.F.64.X	Ensemble multiply floating-point double exact
E.MUL.F.64.Z	Ensemble multiply floating-point double zero
E.MUL.F.128	Ensemble multiply floating-point quad
E.MUL.F.128.C	Ensemble multiply floating-point quad ceiling
E.MUL.F.128.F	Ensemble multiply floating-point quad floor
E.MUL.F.128.N	Ensemble multiply floating-point quad nearest
E.MUL.F.128.X	Ensemble multiply floating-point quad exact
E.MUL.F.128.Z	Ensemble multiply floating-point quad zero

FIG. 30A-2

U.S. Patent

Apr. 20, 2004

Sheet 130 of 148

US 6,725,356 B2

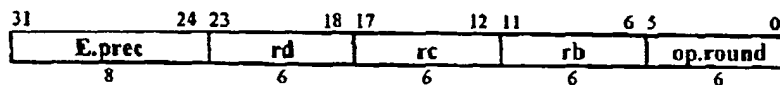
Selection

class	op	prec				round/trap
add	EADDF	16	32	64	128	NONE C F N X Z
divide	EDIVF	16	32	64	128	NONE C F N X Z
multiply	EMULF	16	32	64	128	NONE C F N X Z
complex multiply	EMUL.CF	16	32	64		NONE

Format

E.op.prec.round rd=rc,rb

rd=eopprecround(rc,rb)

**FIG. 30B**

U.S. Patent

Apr. 20, 2004

Sheet 131 of 148

US 6,725,356 B2

Definition

```

def mul(size,v,i,w,j) as
  mul ← fmul(F(size,vsize-1+i..i),F(size,wsizc-1+j..j))
enddef

def EnsembleFloatingPoint(op,prec,round,ra,rb,rc) as
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  for i ← 0 to 128-prec by prec
    ci ← F(prec,ci+prec-1..i)
    bi ← F(prec,bi+prec-1..i)
    case op of
      E.ADD.F:
        ai ← faddr(ci,bi,round)
      E.MUL.F:
        ai ← fmul(ci,bi)
      E.MUL.C.F:
        if (i and prec) then
          ai ← fadd(mul(prec,c,i,b,i-prec), mul(prec,c,i-prec,b,i))
        else
          ai ← fsub(mul(prec,c,i,b,i), mul(prec,c,i+prec,b,i+prec))
        endif
      E.DIV.F.:
        ai ← fdiv(ci,bi)
    endcase
    ai+prec-1..i ← PackF(prec, ai, round)
  endfor
  RegWrite(rd, 128, a)
enddef

```

FIG. 30C

U.S. Patent

Apr. 20, 2004

Sheet 132 of 148

US 6,725,356 B2

Operation codes

E.SUB.F.16	Ensemble subtract floating-point half
E.SUB.F.16.C	Ensemble subtract floating-point half ceiling
E.SUB.F.16.F	Ensemble subtract floating-point half floor
E.SUB.F.16.N	Ensemble subtract floating-point half nearest
E.SUB.F.16.Z	Ensemble subtract floating-point half zero
E.SUB.F.16.X	Ensemble subtract floating-point half exact
E.SUB.F.32	Ensemble subtract floating-point single
E.SUB.F.32.C	Ensemble subtract floating-point single ceiling
E.SUB.F.32.F	Ensemble subtract floating-point single floor
E.SUB.F.32.N	Ensemble subtract floating-point single nearest
E.SUB.F.32.Z	Ensemble subtract floating-point single zero
E.SUB.F.32.X	Ensemble subtract floating-point single exact
E.SUB.F.64	Ensemble subtract floating-point double
E.SUB.F.64.C	Ensemble subtract floating-point double ceiling
E.SUB.F.64.F	Ensemble subtract floating-point double floor
E.SUB.F.64.N	Ensemble subtract floating-point double nearest
E.SUB.F.64.Z	Ensemble subtract floating-point double zero
E.SUB.F.64.X	Ensemble subtract floating-point double exact
E.SUB.F.128	Ensemble subtract floating-point quad
E.SUB.F.128.C	Ensemble subtract floating-point quad ceiling
E.SUB.F.128.F	Ensemble subtract floating-point quad floor
E.SUB.F.128.N	Ensemble subtract floating-point quad nearest
E.SUB.F.128.Z	Ensemble subtract floating-point quad zero
E.SUB.F.128.X	Ensemble subtract floating-point quad exact

FIG. 31A

U.S. Patent

Apr. 20, 2004

Sheet 133 of 148

US 6,725,356 B2

Selection

class	op	prec	round/trap
set	SET. E LG L GE	16 32 64 128	NONE X
subtract	SUB	16 32 64 128	NONE C F N X Z

Format

E.op.prec.round rd=rb,rc

rd=eopprecround(rb,rc)

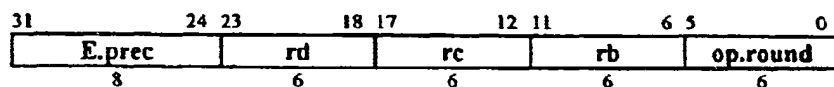


FIG. 31B

U.S. Patent

Apr. 20, 2004

Sheet 134 of 148

US 6,725,356 B2

Definition

```
def EnsembleReversedFloatingPoint(op,prec,round,rd,rc,rb) as
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  for i ← 0 to 128-prec by prec
    ci ← F(prec,ci+prec-1..i)
    bi ← F(prec,bi+prec-1..i)
    ai ← frsubr(ci,-bi, round)
    ai+prec-1..i ← PackF(prec, ai, round)
  endfor
  RegWrite(rd, 128, a)
enddef
```

FIG. 31C

U.S. Patent

Apr. 20, 2004

Sheet 135 of 148

US 6,725,356 B2

Operation codes

X.COMPRESS.2	Crossbar compress signed pecks
X.COMPRESS.4	Crossbar compress signed nibbles
X.COMPRESS.8	Crossbar compress signed bytes
X.COMPRESS.16	Crossbar compress signed doublets
X.COMPRESS.32	Crossbar compress signed quadlets
X.COMPRESS.64	Crossbar compress signed octlets
X.COMPRESS.128	Crossbar compress signed hexlet
X.COMPRESS.U.2	Crossbar compress unsigned pecks
X.COMPRESS.U.4	Crossbar compress unsigned nibbles
X.COMPRESS.U.8	Crossbar compress unsigned bytes
X.COMPRESS.U.16	Crossbar compress unsigned doublets
X.COMPRESS.U.32	Crossbar compress unsigned quadlets
X.COMPRESS.U.64	Crossbar compress unsigned octlets
X.COMPRESS.U.128	Crossbar compress unsigned hexlet
X.EXPAND.2	Crossbar expand signed pecks
X.EXPAND.4	Crossbar expand signed nibbles
X.EXPAND.8	Crossbar expand signed bytes
X.EXPAND.16	Crossbar expand signed doublets
X.EXPAND.32	Crossbar expand signed quadlets
X.EXPAND.64	Crossbar expand signed octlets
X.EXPAND.128	Crossbar expand signed hexlet
X.EXPAND.U.2	Crossbar expand unsigned pecks
X.EXPAND.U.4	Crossbar expand unsigned nibbles
X.EXPAND.U.8	Crossbar expand unsigned bytes
X.EXPAND.U.16	Crossbar expand unsigned doublets
X.EXPAND.U.32	Crossbar expand unsigned quadlets
X.EXPAND.U.64	Crossbar expand unsigned octlets
X.EXPAND.U.128	Crossbar expand unsigned hexlet
X.ROTL.2	Crossbar rotate left pecks
X.ROTL.4	Crossbar rotate left nibbles
X.ROTL.8	Crossbar rotate left bytes
X.ROTL.16	Crossbar rotate left doublets
X.ROTL.32	Crossbar rotate left quadlets
X.ROTL.64	Crossbar rotate left octlets
X.ROTL.128	Crossbar rotate left hexlet
X.ROTR.2	Crossbar rotate right pecks
X.ROTR.4	Crossbar rotate right nibbles
X.ROTR.8	Crossbar rotate right bytes
X.ROTR.16	Crossbar rotate right doublets

FIG. 32A-1

U.S. Patent

Apr. 20, 2004

Sheet 136 of 148

US 6,725,356 B2

X.ROTR.32	Crossbar rotate right quadlets
X.ROTR.64	Crossbar rotate right octlets
X.ROTR.128	Crossbar rotate right hexlet
X.SHL.2	Crossbar shift left pecks
X.SHL.2.O	Crossbar shift left signed pecks check overflow
X.SHL.4	Crossbar shift left nibbles
X.SHL.4.O	Crossbar shift left signed nibbles check overflow
X.SHL.8	Crossbar shift left bytes
X.SHL.8.O	Crossbar shift left signed bytes check overflow
X.SHL.16	Crossbar shift left doublets
X.SHL.16.O	Crossbar shift left signed doublets check overflow
X.SHL.32	Crossbar shift left quadlets
X.SHL.32.O	Crossbar shift left signed quadlets check overflow
X.SHL.64	Crossbar shift left octlets
X.SHL.64.O	Crossbar shift left signed octlets check overflow
X.SHL.128	Crossbar shift left hexlet
X.SHL.128.O	Crossbar shift left signed hexlet check overflow
X.SHL.U.2.O	Crossbar shift left unsigned pecks check overflow
X.SHL.U.4.O	Crossbar shift left unsigned nibbles check overflow
X.SHL.U.8.O	Crossbar shift left unsigned bytes check overflow
X.SHL.U.16.O	Crossbar shift left unsigned doublets check overflow
X.SHL.U.32.O	Crossbar shift left unsigned quadlets check overflow
X.SHL.U.64.O	Crossbar shift left unsigned octlets check overflow
X.SHL.U.128.O	Crossbar shift left unsigned hexlet check overflow
X.SHR.2	Crossbar signed shift right pecks
X.SHR.4	Crossbar signed shift right nibbles
X.SHR.8	Crossbar signed shift right bytes
X.SHR.16	Crossbar signed shift right doublets
X.SHR.32	Crossbar signed shift right quadlets
X.SHR.64	Crossbar signed shift right octlets
X.SHR.128	Crossbar signed shift right hexlet
X.SHR.U.2	Crossbar shift right unsigned pecks
X.SHR.U.4	Crossbar shift right unsigned nibbles
X.SHR.U.8	Crossbar shift right unsigned bytes
X.SHR.U.16	Crossbar shift right unsigned doublets
X.SHR.U.32	Crossbar shift right unsigned quadlets
X.SHR.U.64	Crossbar shift right unsigned octlets
X.SHR.U.128	Crossbar shift right unsigned hexlet

FIG. 32A-2

U.S. Patent

Apr. 20, 2004

Sheet 137 of 148

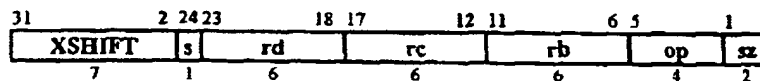
US 6,725,356 B2**Selection**

class	op				size							
precision	EXPAND		EXPAND.U		2	4	8	16	32	64	128	
	COMPRESS		COMPRESS.U									
shift	ROTR	ROTL	SHR	SHL	2	4	8	16	32	64	128	
	SHL.O	SHL.U.O	SHR.U									

Format

X.op.size rd=rc,rb

rd=xopsize(rc,rb)



lsize ← log(size)

s ← lsize₂sz ← lsize_{1..0}**FIG. 32B**

U.S. Patent

Apr. 20, 2004

Sheet 138 of 148

US 6,725,356 B2

Definition

```

def Crossbar(op, size, rd, rc, rb)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  shift ← b and (size-1)
  case op5..2 || 02 of
    X.COMPRESS:
      hsize ← size/2
      for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
          ai+hsize-1..i ← ci+shift+hsize-1..i+shift
        else
          ai+hsize-1..i ← cshift-hsize..i+size-1 || ci+size-1..i+shift
        endif
      endfor
      a127..64 ← 0
    X.COMPRESS.U:
      hsize ← size/2
      for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
          ai+hsize-1..i ← ci+shift+hsize-1..i+shift
        else
          ai+hsize-1..i ← 0shift-hsize || ci+size-1..i+shift
        endif
      endfor
      a127..64 ← 0
    X.EXPAND:
      hsize ← size/2
      for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
          ai+i+size-1..i+i ← chsize-shift..i+hsize-1 || ci+hsize-1..i || 0shift
        else
          ai+i+size-1..i+i ← ci+size-shift-1..i || 0shift
        endif
      endfor
  end

```

FIG. 32C-1

U.S. Patent

Apr. 20, 2004

Sheet 139 of 148

US 6,725,356 B2

X.EXPAND.U:

```

    hsize ← size/2
    for i ← 0 to 64-hsize by hsize
        if shift ≤ hsize then
             $a_{i+i+size-1..i+i} \leftarrow 0_{hsize-shift} \parallel c_{i+hsize-1..i} \parallel 0_{shift}$ 
        else
             $a_{i+i+size-1..i+i} \leftarrow c_{i+size-shift-1..i} \parallel 0_{shift}$ 
        endif
    endfor

```

X.ROTL:

```

    for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \parallel c_{i+size-1..i+size-1-shift}$ 
    endfor

```

X.ROTR:

```

    for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow c_{i+shift-1..i} \parallel c_{i+size-1..i+shift}$ 
    endfor

```

X.SHL:

```

    for i ← 0 to 128-size by size
         $a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \parallel 0_{shift}$ 
    endfor

```

X.SHL.C:

```

    for i ← 0 to 128-size by size
        if  $c_{i+size-1..i+size-1-shift} \neq c_{i+size-1-shift}^{shift+1}$  then
            raise FixedPointArithmetic
        endif
         $a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \parallel 0_{shift}$ 
    endfor

```

FIG. 32C-2

U.S. Patent

Apr. 20, 2004

Sheet 140 of 148

US 6,725,356 B2

```

X.SHL.U.O:
  for i ← 0 to 128-size by size
    if  $c_{i+size-1..i+size-shift} \neq 0^{shift}$  then
      raise FixedPointArithmetic
    endif
     $a_{i+size-1..i} \leftarrow c_{i+size-1-shift..i} \parallel 0^{shift}$ 
  endfor
X.SHR:
  for i ← 0 to 128-size by size
     $a_{i+size-1..i} \leftarrow c_{i+size-1}^{shift} \parallel c_{i+size-1..i+shift}$ 
  endfor
X.SHR.U:
  for i ← 0 to 128-size by size
     $a_{i+size-1..i} \leftarrow 0^{shift} \parallel c_{i+size-1..i+shift}$ 
  endfor
endcase
RegWrite(rd, 128, a)
enddef

```

FIG. 32C -3

U.S. Patent

Apr. 20, 2004

Sheet 141 of 148

US 6,725,356 B2

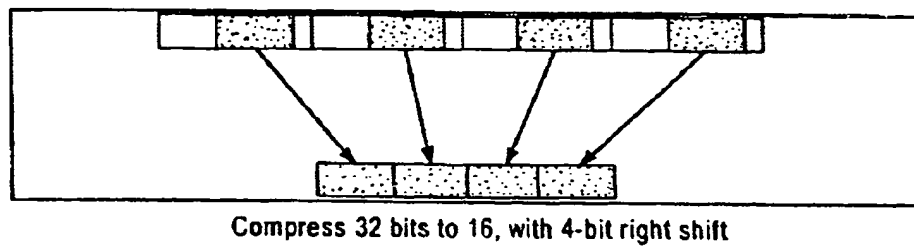


FIG. 32D

U.S. Patent

Apr. 20, 2004

Sheet 142 of 148

US 6,725,356 B2

Format

X.EXTRACT ra=rd,rc,rb

ra=xextract(rd,rc,rb)

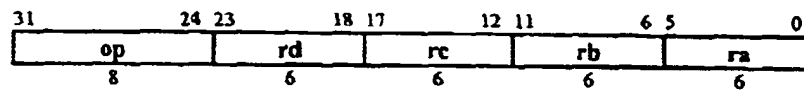


FIG. 33A

U.S. Patent

Apr. 20, 2004

Sheet 143 of 148

US 6,725,356 B2

Definition

```

def CrossbarExtract(op,ra,rb,rc,rd) as
  d ← RegRead(rd, 128)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  case b8..0 of
    0..255:
      gsize ← 128
    256..383:
      gsize ← 64
    384..447:
      gsize ← 32
    448..479:
      gsize ← 16
    480..495:
      gsize ← 8
    496..503:
      gsize ← 4
    504..507:
      gsize ← 2
    508..511:
      gsize ← 1
  endcase
  m ← b12
  as ← signed ← b14
  h ← (2-m)*gsiz
  spos ← (b8..0) and ((2-m)*gsiz-1)
  dpos ← (0 || b23..16) and (gsiz-1)
  sfsiz ← (0 || b31..24) and (gsiz-1)
  tfsiz ← (sfsiz = 0) or ((sfsiz+dpos) > gsiz) ? gsiz-dpos : sfsiz
  fsiz ← (tfsiz + spos > h) ? h - spos : tfsiz
  for i ← 0 to 128-gsiz by gsiz
    case op of
      X.EXTRACT:
        if m then
          p ← dgsiz+i-1..i
        else
          p ← (d || c)2*(gsiz+i)-1..2*i
        endif
      endcase
      v ← (as & ph-1)lp
      w ← (as & vspos+fsiz-1)gsiz-fsiz-dpos || vfsiz-1+spos..spos || 0dpos
      if m then
        asiz-1+i..i ← cgsiz-1+i..dpos+fsiz+i || wdpos+fsiz-1..dpos || cdpos-1+1..i
      else
        asiz-1+i..i ← w
      endif
    endfor
  RegWrite(ra, 128, a)
enddef

```

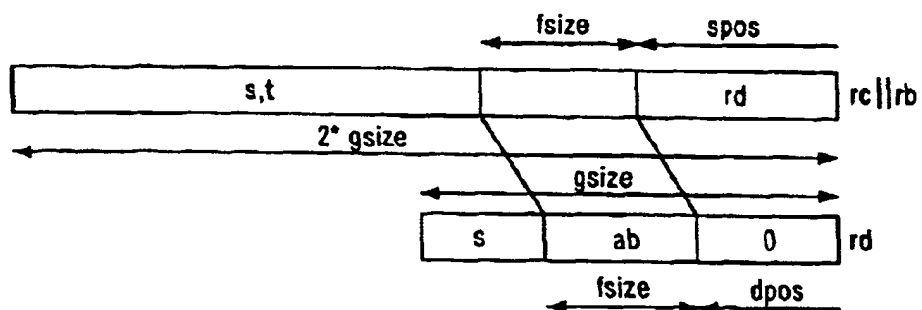
FIG. 33B

U.S. Patent

Apr. 20, 2004

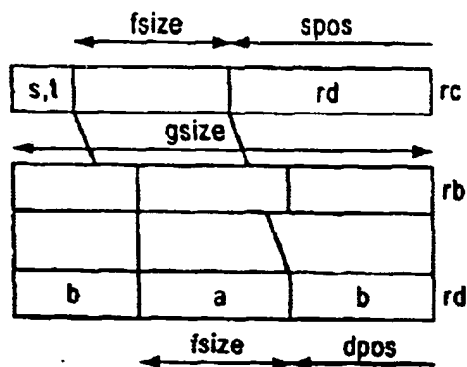
Sheet 144 of 148

US 6,725,356 B2



Crossbar extract

FIG. 33C



Crossbar merge extract

FIG. 33D

U.S. Patent

Apr. 20, 2004

Sheet 145 of 148

US 6,725,356 B2

X.SHUFFLE.4	Crossbar shuffle within pecks
X.SHUFFLE.8	Crossbar shuffle within bytes
X.SHUFFLE.16	Crossbar shuffle within doublets
X.SHUFFLE.32	Crossbar shuffle within quadlets
X.SHUFFLE.64	Crossbar shuffle within octlets
X.SHUFFLE.128	Crossbar shuffle within hexlet
X.SHUFFLE.256	Crossbar shuffle within trilet

FIG. 34A

U.S. Patent

Apr. 20, 2004

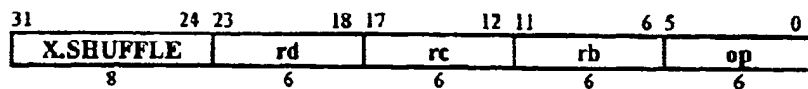
Sheet 146 of 148

US 6,725,356 B2

Format**X.SHUFFLE.256** rd=rc,rb,v,w,h**X.SHUFFLE.size** rd=rcb,v,w

rd=xshuffle256(rc,rb,v,w,h)

rd=xshufflesize(rcb,v,w)

 $rc \leftarrow rb \leftarrow rcb$ $x \leftarrow \log_2(\text{size})$ $y \leftarrow \log_2(v)$ $z \leftarrow \log_2(w)$ $op \leftarrow ((x^3 - 3x^2 + 4x)/6 - (z^2 - z)/2 + xz + y) + (\text{size} = 256) * (h * 32 - 56)$ **FIG. 34B**

U.S. Patent

Apr. 20, 2004

Sheet 147 of 148

US 6,725,356 B2

Definition

```

def CrossbarShuffle(major,rd,rc,rb,op)
  c ← RegRead(rc, 128)
  b ← RegRead(rb, 128)
  if rc=rb then
    case op of
      0..55:
        for x ← 2 to 7; for y ← 0 to x-2; for z ← 1 to x-y-1
          if op = ((x*x*x-3*x*x-4*x)/6-(z*z-z)/2+x*z+y) then
            for i ← 0 to 127
              ai ← c(i6..x || iy+z-1..y || ix-1..y+z || iy-1..0)
            end
          endif
        endfor; endfor; endfor
      56..63:
        raise ReservedInstruction
    endcase
  elseif
    case op4..0 of
      0..27:
        cb ← c || b
        x ← 8
        h ← op5
        for y ← 0 to x-2; for z ← 1 to x-y-1
          if op4..0 = ((17*z-z*z)/2-8+y) then
            for i ← h*128 to 127+h*128
              ai-h*128 ← cb(iy+z-1..y || ix-1..y+z || iy-1..0)
            end
          endif
        endfor; endfor
      28..31:
        raise ReservedInstruction
    endcase
  endif
  RegWrite(rd, 128, a)
enddef

```

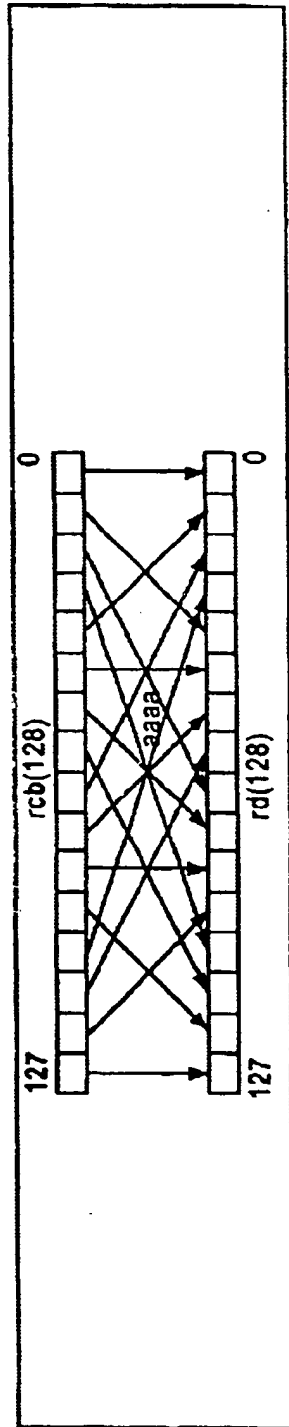
FIG. 34C

U.S. Patent

Apr. 20, 2004

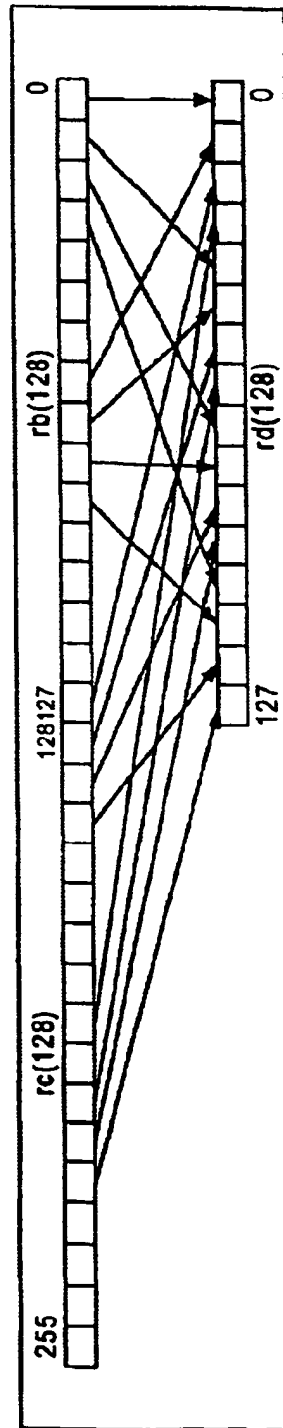
Sheet 148 of 148

US 6,725,356 B2



4-way shuffle bytes within hexlet

FIG. 34D



4-way shuffle bytes within triclet

FIG. 34E

US 6,725,356 B2

1

SYSTEM WITH WIDE OPERAND ARCHITECTURE, AND METHOD

RELATED APPLICATIONS

This application is a continuation of U.S. patent application Ser. No. 09/382,402, filed Aug. 24, 1999, now U.S. Pat. No. 6,295,599, which claims the benefit of priority to Provisional Application No. 60/097,635 filed on Aug. 24, 1998, and which is a continuation-in-part of U.S. patent application Ser. No. 09/169,963, filed Oct. 13, 1998, now U.S. Pat. No. 6,006,318, which is a continuation of U.S. patent application Ser. No. 08/754,827, filed Nov. 22, 1996 now U.S. Pat. No. 5,822,603, which is a divisional of U.S. patent application Ser. No. 08/516,036, filed Aug. 16, 1995 now U.S. Pat. No. 5,742,840.

MICROFICHE APPENDIX

Include herewith as an Appendix are 5 sheets of microfiche of 63 frames each.

FIELD OF THE INVENTION

The present invention relates to general purpose processor architectures, and particularly relates to wide operand architectures.

REFERENCE TO A "SEQUENCE LISTING," A TABLE, OR A COMPUTER PROGRAM LISTING APPENDIX SUBMITTED ON A COMPACT DISK

This application includes an appendix, submitted herewith in duplicate on compact disks labeled as "Copy 1" and "Copy 2." The contents of the compact disks are hereby incorporated by reference.

BACKGROUND OF THE INVENTION

The performance level of a processor, and particularly a general purpose processor, can be estimated from the multiple of a plurality of interdependent factors: clock rate, gates per clock, number of operands, operand and data path width, and operand and data path partitioning. Clock rate is largely influenced by the choice of circuit and logic technology, but is also influenced by the number of gates per clock. Gates per clock is how many gates in a pipeline may change state in a single clock cycle. This can be reduced by inserting latches into the data path: when the number of gates between latches is reduced, a higher clock is possible. However, the additional latches produce a longer pipeline length, and thus come at a cost of increased instruction latency. The number of operands is straightforward; for example, by adding with carry-save techniques, three values may be added together with little more delay than is required for adding two values. Operand and data path width defines how much data can be processed at once; wider data paths can perform more complex functions, but generally this comes at a higher implementation cost. Operand and data path partitioning refers to the efficient use of the data path as width is increased, with the objective of maintaining substantially peak usage.

The last factor, operand and data path partitioning, is treated extensively in commonly-assigned U.S. Pat. Nos. 5,742,840, 5,794,060, 5,794,061, 5,809,321, and 5,822,603, which describe systems and methods for enhancing the utilization of a general purpose processor by adding classes of instructions. These classes of instructions use the contents of general purpose registers as data path sources, partition

2

the operands into symbols of a specified size, perform operations in parallel, concatenate the results and place the concatenated results into a general-purpose register. These patents, all of which are assigned to the same assignee as the present invention, teach a general purpose microprocessor which has been optimized for processing and transmitting media data streams through significant parallelism.

While the foregoing patents offered significant improvements in utilization and performance of a general purpose microprocessor, particularly for handling broadband communications such as media data streams, other improvements are possible.

Many general purpose processors have general registers to store operands for instructions, with the register width matched to the size of the data path. Processor designs generally limit the number of accessible registers per instruction because the hardware to access these registers is relatively expensive in power and area. While the number of accessible registers varies among processor designs, it is often limited to two, three or four registers per instruction when such instructions are designed to operate in a single processor clock cycle or a single pipeline flow. Some processors, such as the Motorola 68000 have instructions to save and restore an unlimited number of registers, but require multiple cycles to perform such an instruction.

The Motorola 68000 also attempts to overcome a narrow data path combined with a narrow register file by taking multiple cycles or pipeline flows to perform an instruction, and thus emulating a wider data path. However, such multiple precision techniques offer only marginal improvement in view of the additional clock cycles required. The width and accessible number of the general purpose registers thus fundamentally limits the amount of processing that can be performed by a single instruction in a register-based machine.

Existing processors may provide instructions that accept operands for which one or more operands are read from a general purpose processor's memory system. However, as these memory operands are generally specified by register operands, and the memory system data path is no wider than the processor data path, the width and accessible number of general purpose operands per instruction per cycle or pipeline flow is not enhanced.

The number of general purpose register operands accessible per instruction is generally limited by logical complexity and instruction size. For example, it might be possible to implement certain desirable but complex functions by specifying a large number of general purpose registers, but substantial additional logic would have to be added to a conventional design to permit simultaneous reading and bypassing of the register values. While dedicated registers have been used in some prior art designs to increase the number or size of source operands or results, explicit instructions load or store values into these dedicated registers, and additional instructions are required to save and restore these registers upon a change of processor context.

There has therefore been a need for a processor system capable of efficient handling of operands of greater width than either the memory system or any accessible general purpose register.

SUMMARY OF THE INVENTION

The present invention provides a system and method for improving the performance of general purpose processors by expanding at least one source operand to a width greater than the width of either the general purpose register or the data

US 6,725,356 B2

3

path width. In addition, several classes of instructions will be provided which cannot be performed efficiently if the operands are limited to the width and accessible number of general purpose registers.

In the present invention, operands are provided which are substantially larger than the data path width of the processor. This is achieved, in part, by using a general purpose register to specify a memory address from which at least more than one, but typically several data path widths of data can be read. To permit such a wide operand to be performed in a single cycle, the data path functional unit is augmented with dedicated storage to which the memory operand is copied on an initial execution of the instruction. Further execution of the instruction or other similar instructions that specify the same memory address can read the dedicated storage to obtain the operand value. However, such reads are subject to conditions to verify that the memory operand has not been altered by intervening instructions. If the memory operand remains current—that is, the conditions are met—the memory operand fetch can be combined with one or more register operands in the functional unit, producing a result. The size of the result is, typically, constrained to that of a general register so that no dedicated or other special storage is required for the result.

Exemplary instructions using wide operations include wide instructions that perform bit level switching (Wide Switch), byte or larger table-lookup (Wide Translate), Wide Multiply Matrix, Wide Multiply Matrix Extract, Wide Multiply Matrix Extract Immediate, Wide Multiply Matrix Floating point, and Wide Multiply Matrix Galois.

Another aspect of the present invention addresses efficient usage of a multiplier array that is fully used for high precision arithmetic, but is only partly used for other, lower precision operations. This can be accomplished by extracting the high-order portion of the multiplier product or sum of products, adjusted by a dynamic shift amount from a general register or an adjustment specified as part of the instruction, and rounded by a control value from a register or instruction portion. The rounding may be any of several types, including round-to-nearest/even, toward zero, floor, or ceiling. Overflows are typically handled by limiting the result to the largest and smallest values that can be accurately represented in the output result.

When an extract is controlled by a register, the size of the result can be specified, allowing rounding and limiting to a smaller number of bits than can fit in the result. This permits the result to be scaled for use in subsequent operations without concern of overflow or rounding. As a result, performance is enhanced. In those instances where the extract is controlled by a register, a single register value defines the size of the operands, the shift amount and size of the result, and the rounding control. By placing such control information in a single register, the size of the instruction is reduced over the number of bits that such an instruction would otherwise require, again improving performance and enhancing processor flexibility. Exemplary instructions are Ensemble Convolve Extract, Ensemble Multiply Extract, Ensemble Multiply Add Extract, and Ensemble Scale Add Extract. With particular regard to the Ensemble Scale Add Extract Instruction, the extract control information is combined in a register with two values used as scalar multipliers to the contents of two vector multiplicands. This combination reduces the number of registers otherwise required, thus reducing the number of bits required for the instruction.

THE FIGURES

FIG. 1 is a system level diagram showing the functional blocks of a system in accordance with an exemplary embodiment of the present invention.

4

FIG. 2 is a matrix representation of a wide matrix multiply in accordance with an exemplary embodiment of the present invention.

FIG. 3 is a further representation of a wide matrix multiply in accordance with an exemplary embodiment of the present invention.

FIG. 4 is a system level diagram showing the functional blocks of a system incorporating a combined Simultaneous Multi Threading and Decoupled Access from Execution processor in accordance with an exemplary embodiment of the present invention.

FIG. 5 illustrates a wide operand in accordance with an exemplary embodiment of the present invention.

FIG. 6 illustrates an approach to specifier decoding in accordance with an exemplary embodiment of the present invention.

FIG. 7 illustrates in operational block form a Wide Function Unit in accordance with an exemplary embodiment of the present invention.

FIG. 8 illustrates in flow diagram form the Wide Micro-cache control function in accordance with an exemplary embodiment of the present invention.

FIG. 9 illustrates Wide Microcache data structures in accordance with an exemplary embodiment of the present invention.

FIGS. 10 and 11 illustrate a Wide Microcache control in accordance with an exemplary embodiment of the present invention.

FIGS. 12A–12D illustrate a Wide Switch instruction in accordance with an exemplary embodiment of the present invention.

FIGS. 13A–13D illustrate a Wide Translate instruction in accordance with an exemplary embodiment of the present invention.

FIGS. 14A–14E illustrate a Wide Multiply Matrix instruction in accordance with an exemplary embodiment of the present invention.

FIGS. 15A–15F illustrate a Wide Multiply Matrix Extract instruction in accordance with an exemplary embodiment of the present invention.

FIGS. 16A–16E illustrate a Wide Multiply Matrix Extract Immediate instruction in accordance with an exemplary embodiment of the present invention.

FIGS. 17A–17E illustrate a Wide Multiply Matrix Floating point instruction in accordance with an exemplary embodiment of the present invention.

FIGS. 18A–18D illustrate a Wide Multiply Matrix Galois instruction in accordance with an exemplary embodiment of the present invention.

FIGS. 19A–19G illustrate an Ensemble Extract Inplace instruction in accordance with an exemplary embodiment of the present invention.

FIGS. 20A–20I illustrate an Ensemble Extract instruction in accordance with an exemplary embodiment of the present invention.

FIGS. 21A–21F illustrate a System and Privileged Library Calls in accordance with an exemplary embodiment of the present invention.

FIGS. 22A–22B illustrate an Ensemble Scale-Add Floating-point instruction in accordance with an exemplary embodiment of the present invention.

FIGS. 23A–23C illustrate a Group Boolean instruction in accordance with an exemplary embodiment of the present invention.

US 6,725,356 B2

5

FIGS. 24A–24C illustrate a Branch Hint instruction in accordance with an exemplary embodiment of the present invention.

FIGS. 25A–25D illustrate an Ensemble Sink Floating-point instruction in accordance with an exemplary embodiment of the present invention.

FIGS. 26A–26C illustrate Group Add instructions in accordance with an exemplary embodiment of the present invention.

FIGS. 27A–27C illustrate Group Set instructions and Group Subtract instructions in accordance with an exemplary embodiment of the present invention.

FIGS. 28A–28C illustrate Ensemble Convolve, Ensemble Divide, Ensemble Multiply, and Ensemble Multiply Sum instructions in accordance with an exemplary embodiment of the present invention.

FIG. 29 illustrates exemplary functions that are defined for use within the detailed instruction definitions in other sections.

FIGS. 30A–30C illustrate Ensemble Floating-Point Add, Ensemble Floating-Point Divide, and Ensemble Floating-Point Multiply instructions in accordance with an exemplary embodiment of the present invention.

FIGS. 31A–31C illustrate Ensemble Floating-Point Subtract instructions in accordance with an exemplary embodiment of the present invention.

FIGS. 32A–32D illustrate Crossbar Compress, Expand, Rotate, and Shift instructions in accordance with an exemplary embodiment of the present invention.

FIGS. 33A–33D illustrate Extract instructions in accordance with an exemplary embodiment of the present invention.

FIGS. 34A–34E illustrate Shuffle instructions in accordance with an exemplary embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

Processor Layout

Referring first to FIG. 1, a general purpose processor is illustrated therein in block diagram form. In FIG. 1, four copies of an access unit are shown, each with an access instruction fetch queue A-Queue 101–104. Each access instruction fetch queue A-Queue 101–104 is coupled to an access register file AR 105–108, which are each coupled to two access functional units A 109–116. In a typical embodiment, each thread of the processor may have on the order of sixty-four general purpose registers (e.g., the AR's 105–108 and ER's 125–128). The access units function independently for four simultaneous threads of execution, and each compute program control flow by performing arithmetic and branch instructions and access memory by performing load and store instructions. These access units also provide wide operand specifiers for wide operand instructions. These eight access functional units A 109–116 produce results for access register files AR 105–108 and memory addresses to a shared memory system 117–120.

In one embodiment, the memory hierarchy includes on-chip instruction and data memories, instruction and data caches, a virtual memory facility, and interfaces to external devices. In FIG. 1, the memory system is comprised of a combined cache and niche memory 117, an external bus interface 118, and, externally to the device, a secondary cache 119 and main memory system with I/O devices 120.

6

The memory contents fetched from memory system 117–120, are combined with execute instructions not performed by the access unit, and entered into the four execute instruction queues E-Queue 121–124. For wide instructions, memory contents fetched from memory system 117–120 are also provided to wide operand microcaches 132–136 by bus 137. Instructions and memory data from E-queue 121–124 are presented to execution register files 125–128, which fetch execution register file source operands. The instructions are coupled to the execution unit arbitration unit Arbitration 131, that selects which instructions from the four threads are to be routed to the available execution functional units E 141 and 149, X 142 and 148, G 143–144 and 146–147, and T 145. The execution functional units E 141 and 149, the execution functional units X 142 and 148, and the execution functional unit T 145 each contain a wide operand microcache 132–136, which are each coupled to the memory system 117 by bus 137.

The execution functional units G 143–144 and 146–147 are group arithmetic and logical units that perform simple arithmetic and logical instructions, including group operations wherein the source and result operands represent a group of values of a specified symbol size, which are partitioned and operated on separately, with results concatenated together. In a presently preferred embodiment the data path is 128 bits wide, although the present invention is not intended to be limited to any specific size of data path.

The execution functional units X 142 and 148 are crossbar switch units that perform crossbar switch instructions. The crossbar switch units 142 and 148 perform data handling operations on the data stream provided over the data path source operand buses 151–158, including deals, shuffles, shifts, expands, compresses, swizzles, permutes and reverses, plus the wide operations discussed hereinafter. In a key element of a first aspect of the invention, at least one such operation will be expanded to a width greater than the general register and data path width.

The execution functional units E 141 and 149 are ensemble units that perform ensemble instructions using a large array multiplier, including group or vector multiply and matrix multiply of operands partitioned from data path source operand buses 151–158 and treated as integer, floating point, polynomial or Galois field values. Matrix multiply instructions and other operations utilize a wide operand loaded into the wide operand microcache 132 and 136.

The execution functional unit T 145 is a translate unit that performs table-look-up operations on a group of operands partitioned from a register operand, and concatenates the result. The Wide Translate instruction utilizes a wide operand loaded into the wide operand microcache 134.

The execution functional units E 141, 149, execution functional units X—142, 148, and execution functional unit T each contain dedicated storage to permit storage of source operands including wide operands as discussed hereinafter. The dedicated storage 132–136, which may be thought of as a wide microcache, typically has a width which is a multiple of the width of the data path operands related to the data path source operand buses 151–158. Thus, if the width of the data path 151–158 is 128 bits, the dedicated storage 132–136 may have a width of 256, 512, 1024 or 2048 bits. Operands which utilize the full width of the dedicated storage are referred to herein as wide operands, although it is not necessary in all instances that a wide operand use the entirety of the width of the dedicated storage; it is sufficient that the wide operand use a portion greater than the width of the memory data path of the output of the memory system

US 6,725,356 B2

7

117–120 and the functional unit data path of the input of the execution functional units 141–149, though not necessarily greater than the width of the two combined. Because the width of the dedicated storage 132–136 is greater than the width of the memory operand bus 137, portions of wide operands are loaded sequentially into the dedicated storage 132–136. However, once loaded, the wide operands may then be used at substantially the same time. It can be seen that functional units 141–149 and associated execution registers 125–128 form a data functional unit, the exact elements of which may vary with implementation.

The execution register file ER 125–128 source operands are coupled to the execution units 141–145 using source operand buses 151–154 and to the execution units 145–149 using source operand buses 155–158. The function unit result operands from execution units 141–145 are coupled to the execution register file ER 125–128 using result bus 161 and the function units result operands from execution units 145–149 are coupled to the execution register file using result bus 162.

Wide Multiply Matrix

The wide operands of the present invention provide the ability to execute complex instructions such as the wide multiply matrix instruction shown in FIG. 2, which can be appreciated in an alternative form, as well, from FIG. 3. As can be appreciated from FIGS. 2 and 3, a wide operand permits, for example, the matrix multiplication of various sizes and shapes which exceed the data path width. The example of FIG. 2 involves a matrix specified by register rc having 128*64/size bits (512 bits for this example) multiplied by a vector contained in register rb having 128 bits, to yield a result, placed in register rd, of 128 bits.

The notation used in FIG. 2 and following similar figures illustrates a multiplication as a shaded area at the intersection of two operands projected in the horizontal and vertical dimensions. A summing node is illustrated as a line segment connecting a darkened dot at the location of multiplier products that are summed. Products that are subtracted at the summing node are indicated with a minus symbol within the shaded area.

When the instruction operates on floating-point values, the multiplications and summations illustrated are floating point multiplications and summations. An exemplary embodiment may perform these operations without rounding the intermediate results, thus computing the final result as if computed to infinite precision and then rounded only once.

It can be appreciated that an exemplary embodiment of the multipliers may compute the product in carry-save form and may encode the multiplier rb using Booth encoding to minimize circuit area and delay. It can be appreciated that an exemplary embodiment of such summing nodes may perform the summation of the products in any order, with particular attention to minimizing computation delay, such as by performing the additions in a binary or higher-radix tree, and may use carry-save adders to perform the addition to minimize the summation delay. It can also be appreciated that an exemplary embodiment may perform the summation using sufficient intermediate precision that no fixed-point or floating-point overflows occur on intermediate results.

A comparison of FIGS. 2 and 3 can be used to clarify the relation between the notation used in FIG. 2 and the more conventional schematic notation in FIG. 3, as the same operation is illustrated in these two figures.

Wide Operand

The operands that are substantially larger than the data path width of the processor are provided by using a general-

8

purpose register to specify a memory specifier from which more than one but in some embodiments several data path widths of data can be read into the dedicated storage. The memory specifier typically includes the memory address together with the size and shape of the matrix of data being operated on. The memory specifier or wide operand specifier can be better appreciated from FIG. 5, in which a specifier 500 is seen to be an address, plus a field representative of the size/2 and a further field representative of width/2, where size is the product of the depth and width of the data. The address is aligned to a specified size, for example sixty four bytes, so that a plurality of low order bits (for example, six bits) are zero. The specifier 500 can thus be seen to comprise a first field 505 for the address, plus two field indicia 510 within the low order six bits to indicate size and width.

Specifier Decoding

The decoding of the specifier 500 may be further appreciated from FIG. 6 where, for a given specifier 600 made up of an address field 605 together with a field 610 comprising plurality of low order bits. By a series of arithmetic operations shown at steps 615 and 620, the portion of the field 610 representative of width/2 is developed. In a similar series of steps shown at 625 and 630, the value of t is decoded, which can then be used to decode both size and address. The portion of the field 610 representative of size/2 is decoded as shown at steps 635 and 640, while the address is decoded in a similar way at steps 645 and 650.

Wide Function Unit

The wide function unit may be better appreciated from FIG. 7, in which a register number 700 is provided to an operand checker 705. Wide operand specifier 710 communicates with the operand checker 705 and also addresses memory 715 having a defined memory width. The memory address includes a plurality of register operands 720A n, which are accumulated in a dedicated storage portion 714 of a data functional unit 725. In the exemplary embodiment shown in FIG. 7, the dedicated storage 714 can be seen to have a width equal to eight data path widths, such that eight wide operand portions 730A–H are sequentially loaded into the dedicated storage to form the wide operand. Although eight portions are shown in FIG. 7, the present invention is not limited to eight or any other specific multiple of data path widths. Once the wide operand portions 730A–H are sequentially loaded, they may be used as a single wide operand 735 by the functional element 740, which may be any element(s) from FIG. 1 connected thereto. The result of the wide operand is then provided to a result register 745, which in a presently preferred embodiment is of the same width as the memory width.

Once the wide operand is successfully loaded into the dedicated storage 714, a second aspect of the present invention may be appreciated. Further execution of this instruction or other similar instructions that specify the same memory address can read the dedicated storage to obtain the operand value under specific conditions that determine whether the memory operand has been altered by intervening instructions. Assuming that these conditions are met, the memory operand fetch from the dedicated storage is combined with one or more register operands in the functional unit, producing a result. In some embodiments, the size of the result is limited to that of a general register, so that no similar dedicated storage is required for the result. However, in some different embodiments, the result may be a wide operand, to further enhance performance.

US 6,725,356 B2

9

To permit the wide operand value to be addressed by subsequent instructions specifying the same memory address, various conditions must be checked and confirmed:

Those conditions include:

1. Each memory store instruction checks the memory address against the memory addresses recorded for the dedicated storage. Any match causes the storage to be marked invalid, since a memory store instruction directed to any of the memory addresses stored in dedicated storage 714 means that data has been over-written.
2. The register number used to address the storage is recorded. If no intervening instructions have written to the register, and the same register is used on the subsequent instruction, the storage is valid (unless marked invalid by rule #1).
3. If the register has been modified or a different register number is used, the value of the register is read and compared against the address recorded for the dedicated storage. This uses more resources than #1 because of the need to fetch the register contents and because the width of the register is greater than that of the register number itself. If the address matches, the storage is valid. The new register number is recorded for the dedicated storage.
4. If conditions #2 or #3 are not met, the register contents are used to address the general-purpose processor's memory and load the dedicated storage. If dedicated storage is already fully loaded, a portion of the dedicated storage must be discarded (victimized) to make room for the new value. The instruction is then performed using the newly updated dedicated storage. The address and register number is recorded for the dedicated storage.

By checking the above conditions, the need for saving and restoring the dedicated storage is eliminated. In addition, if the context of the processor is changed and the new context does not employ Wide instructions that reference the same dedicated storage, when the original context is restored, the contents of the dedicated storage are allowed to be used without refreshing the value from memory, using checking rule #3. Because the values in the dedicated storage are read from memory and not modified directly by performing wide operations, the values can be discarded at any time without saving the results into general memory. This property simplifies the implementation of rule #4 above.

An alternate embodiment of the present invention can replace rule #1 above with the following rule:

- 1a. Each memory store instruction checks the memory address against the memory addresses, recorded for the dedicated storage. Any match causes the dedicated storage to be updated, as well as the general memory.

By use of the above rule 1.a, memory store instructions can modify the dedicated storage, updating just the piece of the dedicated storage that has been changed, leaving the remainder intact. By continuing to update the general memory, it is still true that the contents of the dedicated memory can be discarded at any time without saving the results into general memory. Thus rule #4 is not made more complicated by this choice. The advantage of this alternate embodiment is that the dedicated storage need not be discarded (invalidated) by memory store operations.

Wide Microcache Data Structures

Referring next to FIG. 9, an exemplary arrangement of the data structures of the wide microcache or dedicated storage 114 may be better appreciated. The wide microcache

10

contents, wmc.c, can be seen to form a plurality of data path widths 900A-n, although in the example shown the number is eight. The physical address, wmc.pa, is shown as 64 bits in the example shown, although the invention is not limited to a specific width. The size of the contents, wmc.size, is also provided in a field which is shown as 10 bits in an exemplary embodiment. A "contents valid" flag, wmc.cv, of one bit is also included in the data structure, together with a two bit field for thread last used, or wmc.th. In addition, a six bit field for register last used, wmc.reg, is provided in an exemplary embodiment. Further, a one bit flag for register and thread valid, or wmc.rtv, may be provided.

Wide Microcache Control—Software

The process by which the microcache is initially written with a wide operand, and thereafter verified as valid for fast subsequent operations, may be better appreciated from FIG. 8. The process begins at 800, and progresses to step 805 where a check of the register contents is made against the stored value wmc.rc. If true, a check is made at step 810 to verify the thread. If true, the process then advances to step 815 to verify whether the register and thread are valid. If step 815 reports as true, a check is made at step 820 to verify whether the contents are valid. If all of steps 805 through 820 return as true, the subsequent instruction is able to utilize the existing wide operand as shown at step 825, after which the process ends. However, if any of steps 805 through 820 return as false, the process branches to step 830, where content, physical address and size are set. Because steps 805 through 820 all lead to either step 825 or 830, steps 805 through 820 may be performed in any order or simultaneously without altering the process. The process then advances to step 835 where size is checked. This check basically ensures that the size of the translation unit is greater than or equal to the size of the wide operand, so that a physical address can directly replace the use of a virtual address. The concern is that, in some embodiments, the wide operands may be larger than the minimum region that the virtual memory system is capable of mapping. As a result, it would be possible for a single contiguous virtual address range to be mapped into multiple, disjoint physical address ranges, complicating the task of comparing physical addresses. By determining the size of the wide operand and comparing that size against the size of the virtual address mapping region which is referenced, the instruction is aborted with an exception trap if the wide operand is larger than the mapping region. This ensures secure operation of the processor. Software can then re-map the region using a larger size map to continue execution if desired. Thus, if size is reported as unacceptable at step 835, an exception is generated at step 840. If size is acceptable, the process advances to step 845 where physical address is checked. If the check reports as met, the process advances to step 850, where a check of the contents valid flag is made. If either check at step 845 or 850 reports as false, the process branches and new content is written into the dedicated storage 114, with the fields thereof being set accordingly. Whether the check at step 850 reported true, or whether new content was written at step 855, the process advances to step 860 where appropriate fields are set to indicate the validity of the data, after which the requested function can be performed at step 825. The process then ends.

Wide Microcache Control—Hardware

Referring next to FIGS. 10 and 11, which together show the operation of the microcache controller from a hardware

US 6,725,356 B2

11

standpoint, the operation of the microcache controller may be better understood. In the hardware implementation, it is clear that conditions which are indicated as sequential steps in FIG. 8 and 9 above can be performed in parallel, reducing the delay for such wide operand checking. Further, a copy of the indicated hardware may be included for each wide microcache, and thereby all such microcaches as may be alternatively referenced by an instruction can be tested in parallel. It is believed that no further discussion of FIGS. 10 and 11 is required in view of the extensive discussion of FIGS. 8 and 9, above.

Various alternatives to the foregoing approach do exist for the use of wide operands, including an implementation in which a single instruction can accept two wide operands, partition the operands into symbols, multiply corresponding symbols together, and add the products to produce a single scalar value or a vector of partitioned values of width of the register file, possibly after extraction of a portion of the sums. Such an instruction can be valuable for detection of motion or estimation of motion in video compression. A further enhancement of such an instruction can incrementally update the dedicated storage if the address of one wide operand is within the range of previously specified wide operands in the dedicated storage, by loading only the portion not already within the range and shifting the in-range portion as required. Such an enhancement allows the operation to be performed over a "sliding window" of possible values. In such an instruction, one wide operand is aligned and supplies the size and shape information, while the second wide operand, updated incrementally, is not aligned.

Another alternative embodiment of the present invention can define additional instructions where the result operand is a wide operand. Such an enhancement removes the limit that a result can be no larger than the size of a general register, further enhancing performance. These wide results can be cached locally to the functional unit that created them, but must be copied to the general memory system before the storage can be reused and before the virtual memory system alters the mapping of the address of the wide result. Data paths must be added so that load operations and other wide operations can read these wide results—forwarding of a wide result from the output of a functional unit back to its input is relatively easy, but additional data paths may have to be introduced if it is desired to forward wide results back to other functional units as wide operands.

As previously discussed, a specification of the size and shape of the memory operand is included in the low-order bits of the address. In a presently preferred implementation, such memory operands are typically a power of two in size and aligned to that size. Generally, one half the total size is added (or inclusively or'ed, or exclusively or'ed) to the memory address, and one half of the data width is added (or inclusively or'ed, or exclusively or'ed) to the memory address. These bits can be decoded and stripped from the memory address, so that the controller is made to step through all the required addresses. This decreases the number of distinct operands required for these instructions, as the size, shape and address of the memory operand are combined into a single register operand value.

The following table illustrates the arithmetic and descriptive notation used in the pseudocode in the Figures referenced hereinafter:

12

$x + y$	two's complement addition of x and y . Result is the same size as the operands, and operands must be of equal size.
$x - y$	two's complement subtraction of y from x . Result is the same size as the operands, and operands must be of equal size.
$x * y$	two's complement multiplication of x and y . Result is the same size as the operands, and operands must be of equal size.
x / y	two's complement division of x by y . Result is the same size as the operands, and operands must be of equal size.
$x \& y$	bitwise and of x and y . Result is same size as the operands, and operands must be of equal size.
$x y$	bitwise or of x and y . Result is same size as the operands, and operands must be of equal size.
$x \wedge y$	bitwise exclusive-of of x and y . Result is same size as the operands, and operands must be of equal size.
$\sim x$	bitwise inversion of x . Result is same size as the operand.
$x = y$	two's complement equality comparison between x and y . Result is a single bit, and operands must be of equal size.
$x > y$	two's complement inequality comparison between x and y . Result is a single bit, and operands must be of equal size.
$x < y$	two's complement less than comparison between x and y . Result is a single bit, and operands must be of equal size.
$x \geq y$	two's complement greater than or equal comparison between x and y . Result is a single bit, and operands must be of equal size.
\sqrt{x}	floating-point square root of x
$x \cdot y$	concatenation of bit field x to left of bit field y
x^y	binary digit x repeated, concatenated y times. Size of result is y .
x_y	extraction of bit y (using little-endian bit numbering) from value x . Result is a single bit.
$x_{y..z}$	extraction of bit field formed from bits y through z of value x . Size of result is $-z + 1$; if $z > y$, result is an empty string.
$x ? y : z$	value of y , if x is true, otherwise value of z . Value of x is a single bit.
$x \leftarrow y$	bitwise assignment of x to value of y
$x.y$	subfield of structured bitfield x
S_n	signed, two's complement, binary data format of n bytes
U_n	unsigned binary data format of n bytes
F_n	floating-point data format of n bytes

Wide Operations

Particular examples of wide operations which are defined by the present invention include the Wide Switch instruction that performs bit-level switching; the Wide Translate instruction which performs byte (or larger) table lookup; Wide Multiply Matrix; Wide Multiply Matrix Extract and Wide Multiply Matrix Extract Immediate (discussed below), Wide Multiply Matrix Floating-point, and Wide Multiply Matrix Galois (also discussed below). While the discussion below focuses on particular sizes for the exemplary instructions, it will be appreciated that the invention is not limited to a particular width.

Wide Switch

An exemplary embodiment of the Wide Switch instruction is shown in FIGS. 12A–12D. In an exemplary embodiment, the Wide Switch instruction rearranges the contents of up to two registers (256 bits) at the bit level, producing a full-width (128 bits) register result. To control the rearrangement, a wide operand specified by a single register, consisting of eight bits per bit position is used. For each result bit position, eight wide operand bits for each bit position select which of the 256 possible source register bits to place in the result. When a wide operand size smaller than 128 bytes is specified, the high order bits of the memory operand are replaced with values corresponding to the result bit position, so that the memory operand specifies a bit selection within symbols of the operand size, performing the same operation on each symbol.

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from

US 6,725,356 B2

13

memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and concatenate the results together, placing the result in a general register. An exemplary embodiment of the format 1210 of the Wide Switch instruction is shown in FIG. 12A.

An exemplary embodiment of a schematic 1230 of the Wide Switch instruction is shown in FIG. 12B. In an exemplary embodiment, the contents of register rc specifies a virtual address and optionally an operand size, and a value of specified size is loaded from memory. A second value is the concatenated contents of registers rd and rb. Eight corresponding bits from the memory value are used to select a single result bit from the second value, for each corresponding bit position. The group of results is concatenated and placed in register ra.

In an exemplary embodiment, the virtual address must either be aligned to 128 bytes, or must be the sum of an aligned address and one-half of the size of the memory operand in bytes. An aligned address must be an exact multiple of the size expressed in bytes. The size of the memory operand must be 8, 16, 32, 64, or 128 bytes. If the address is not valid an "access disallowed by virtual address" exception occurs. When a size smaller than 128 bits is specified, the high order bits of the memory operand are replaced with values corresponding to the bit position, so that the same memory operand specifies a bit selection within symbols of the operand size, and the same operation is performed on each symbol.

In an exemplary embodiment, a wide switch (W.SWITCH.L or W.SWITCH.B) instruction specifies an 8-bit location for each result bit from the memory operand, that selects one of the 256 bits represented by the concatenated contents of registers rd and rb.

An exemplary embodiment of the pseudocode 1250 of the Wide Switch instruction is shown in FIG. 12C. An exemplary embodiment of the exceptions 1280 of the Wide Switch instruction is shown in FIG. 12D.

Wide Translate

An exemplary embodiment of the Wide Translate instruction is shown in FIGS. 13A-13D. In an exemplary embodiment, the Wide Translate instructions use a wide operand to specify a table of depth up to 256 entries and width of up to 128 bits. The contents of a register is partitioned into operands of one, two, four, or eight bytes, and the partitions are used to select values from the table in parallel. The depth and width of the table can be selected by specifying the size and shape of the wide operand as described above.

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and concatenate the results together, placing the result in a general register. An exemplary embodiment of the format 1310 of the Wide Translate instruction is shown in FIG. 13A.

An exemplary embodiment of the schematic 1330 of the Wide Translate instruction is shown in FIG. 13B. In an exemplary embodiment, the contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rb. The values are partitioned into groups of operands of a size specified. The low-order bytes of the second group of values are used as addresses to choose entries from one or more tables constructed from the first value, producing a group of values. The group of results is concatenated and placed in register rd.

14

In an exemplary embodiment, by default, the total width of tables is 128 bits, and a total table width of 128, 64, 32, 16 or 8 bits, but not less than the group size may be specified by adding the desired total table width in bytes to the specified address: 16, 8, 4, 2, or 1. When fewer than 128 bits are specified, the tables repeat to fill the 128 bit width.

In an exemplary embodiment, the default depth of each table is 256 entries, or in bytes is 32 times the group size in bits. An operation may specify 4, 8, 16, 32, 64, 128 or 256 entry tables, by adding one half of the memory operand size to the address. Table index values are masked to ensure that only the specified portion of the table is used. Tables with just 2 entries cannot be specified; if 2-entry tables are desired, it is recommended to load the entries into registers and use G.MUX to select the table entries.

In an exemplary embodiment, failing to initialize the entire table is a potential security hole, as an instruction in with a small-depth table could access table entries previously initialized by an instruction with a large-depth table. This security hole may be closed either by initializing the entire table, even if extra cycles are required, or by masking the index bits so that only the initialized portion of the table is used. An exemplary embodiment may initialize the entire table with no penalty in cycles by writing to as many as 128 table entries at once. Initializing the entire table with writes to only one entry at a time requires writing 256 cycles, even when the table is smaller. Masking the index bits is the preferred solution.

In an exemplary embodiment, masking the index bits suggests that this instruction, for tables larger than 256 entries, may be extended to a general-purpose memory translate function where the processor performs enough independent load operations to fill the 128 bits. Thus, the 16, 32, and 64 bit versions of this function perform equivalent of 8, 4, 2 withdraw, 8, 4, or 2 load-indexed and 7, 3, or 1 group-extract instructions. In other words, this instruction can be as powerful as 23, 11, or 5 previously existing instructions. The 8-bit version is a single cycle operation replacing 47 existing instructions, so these extensions are not as powerful, but nonetheless, this is at least a 50% improvement on a 2-issue processor, even with one cycle per load timing. To make this possible, the default table size would become 65536, 2³² and 2⁶⁴ for 16, 32 and 64-bit versions of the instruction.

In an exemplary embodiment, for the big-endian version of this instruction, in the definition below, the contents of register rb is complemented. This reflects a desire to organize the table so that the lowest addressed table entries are selected when the index is zero. In the logical implementation, complementing the index can be avoided by loading the table memory differently for big-endian and little-endian versions; specifically by loading the table into memory so that the highest-addressed table entries are selected when the index is zero for a big-endian version of the instruction. In an exemplary embodiment of the logical implementation, complementing the index can be avoided by loading the table memory differently for big-endian and little-endian versions. In order to avoid complementing the index, the table memory is loaded differently for big-endian versions of the instruction by complementing the addresses at which table entries are written into the table for a big-endian version of the instruction.

In an exemplary embodiment, the virtual address must either be aligned to 4096 bytes, or must be the sum of an aligned address and one-half of the size of the memory operand in bytes and/or the desired total table width in bytes.

US 6,725,356 B2

15

An aligned address must be an exact multiple of the size expressed in bytes. The size of the memory operand must be a power of two from 4 to 4096 bytes, but must be at least 4 times the group size and 4 times the total table width. If the address is not valid an "access disallowed by virtual address" exception occurs.

In an exemplary embodiment, a wide translate (W.TRANSLATE.8.L or W.TRANSLATE.8.B) instruction specifies a translation table of 16 entries (vsize=16) in depth, a group size of 1 byte (gsize=8 bits), and a width of 8 bytes (wsize=64 bits). The address specifies a total table size (msize=1024 bits=vsize*wsize) and a table width (wsize=64 bits) by adding one half of the size in bytes of the table (64) and adding the size in bytes of the table width (8) to the table address in the address specification. The instruction will create duplicates of this table in the upper and lower 64 bits of the data path, so that 128 bits of operand are processed at once, yielding a 128 bit result.

An exemplary embodiment of the pseudocode 1350 of the Wide Translate instruction is shown in FIG. 13C. An exemplary embodiment of the exceptions 1380 of the Wide Translate instruction is shown in FIG. 13D.

Wide Multiply Matrix

An exemplary embodiment of the Wide Multiply Matrix instruction is shown in FIGS. 14A–14E. In an exemplary embodiment, the Wide Multiply Matrix instructions use a wide operand to specify a matrix of values of width up to 64 bits (one half of register file and data path width) and depth of up to 128 bits/symbol size. The contents of a general register (128 bits) is used as a source operand, partitioned into a vector of symbols, and multiplied with the matrix, producing a vector of width up to 128 bits of symbols of twice the size of the source operand symbols. The width and depth of the matrix can be selected by specifying the size and shape of the wide operand as described above. Controls within the instruction allow specification of signed, mixed signed, unsigned, complex, or polynomial operands.

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and concatenate the results together, placing the result in a general register. An exemplary embodiment of the format 1410 of the Wide Multiply Matrix instruction is shown in FIG. 14A.

An exemplary embodiment of the schematics 1430 and 1460 of the Wide Multiply Matrix instruction is shown in FIGS. 14B and 14C. In an exemplary embodiment, the contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified. The second values are multiplied with the first values, then summed, producing a group of result values. The group of result values is concatenated and placed in register rd.

In an exemplary embodiment, the memory multiply instructions (W.MUL.MAT.C, W.MUL.MAT.C, W.MUL.MAT.M, W.MUL.MAT.P, W.MUL.MAT.U) perform a partitioned array multiply of up to 8192 bits, that is 64×128 bits. The width of the array can be limited to 64, 32, or 16 bits, but not smaller than twice the group size, by adding one half the desired size in bytes to the virtual address operand: 4, 2, or 1. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired memory operand size in bytes to the virtual address operand.

16

In an exemplary embodiment, the virtual address must either be aligned to 1024/gsize bytes (or 512/gsize for W.MUL.MAT.C) (with gsize measured in bits), or must be the sum of an aligned address and one half of the size of the memory operand in bytes and/or one quarter of the size of the result in bytes. An aligned address must be an exact multiple of the size expressed in bytes. If the address is not valid an "access disallowed by virtual address" exception occurs.

In an exemplary embodiment, a wide multiply octlets instruction (W.MUL.MAT.type.64, type=NONE M U P) is not implemented, and causes a reserved instruction exception, as an ensemble-multiply-sum-octlets instruction (E.MUL.SUM.type.64) performs the same operation except that the multiplier is sourced from a 128-bit register rather than memory. Similarly, instead of wide-multiply-complex-quadtlets instruction (W.MUL.MAT.C.32), one should use an ensemble-multiply-complex-quadtlets instruction (E.MUL.SUM.C.32).

As shown in FIG. 14B, an exemplary embodiment of a wide-multiply-doubles instruction (W.MUL.MAT, W.MUL.MAT.M, W.MUL.MAT.P, W.MUL.MAT.U) multiplies memory [m31 m30 . . . m1 m0] with vector [h g f e d c b a], yielding products [hm31+gm27+ . . . +bm7+am3 . . . hm28+gm24+ . . . +bm4+am0].

As shown in FIG. 14C, an exemplary embodiment of a wide-multiply-matrix-complex-doubles instruction (W.MUL.MAT.C) multiplies memory [m15 m14 . . . m1 m0] with vector [h g f e d c b a], yielding products [hm14+gm15+ . . . +bm2+am3 . . . hm12+gm13+ . . . +bm0+am1 hm13+gm12+ . . . bm1+am0].

An exemplary embodiment of the pseudocode 1480 of the Wide Multiply Matrix instruction is shown in FIG. 14D. An exemplary embodiment of the exceptions 1490 of the Wide Multiply Matrix instruction is shown in FIG. 14E.

Wide Multiply Matrix Extract

An exemplary embodiment of the Wide Multiply Matrix Extract instruction is shown in FIGS. 15A–15F. In an exemplary embodiment, the Wide Multiply Matrix Extract instructions use a wide operand to specify a matrix of value of width up to 128 bits (full width of register file and data path) and depth of up to 128 bits/symbol size. The contents of a general register (128 bits) is used as a source operand, partitioned into a vector of symbols, and multiplied with the matrix, producing a vector of width up to 256 bits of symbols of twice the size of the source operand symbols plus additional bits to represent the sums of products without overflow. The results are then extracted in a manner described below (Enhanced Multiply Bandwidth by Result Extraction), as controlled by the contents of a general register specified by the instruction. The general register also specifies the format of the operands: signed, mixed-signed, unsigned, and complex as well as the size of the operands, byte (8 bit), doublet (16 bit), quadlet (32 bit), or hexlet (64 bit).

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and concatenate the results together, placing the result in a general register. An exemplary embodiment of the format 1510 of the Wide Multiply Matrix Extract instruction is shown in FIG. 15A.

An exemplary embodiment of the schematics 1530 and 1560 of the Wide Multiply Matrix Extract instruction is

US 6,725,356 B2

17

shown in FIGS. 15C and 14D. In an exemplary embodiment, the contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rd. The group size and other parameters are specified from the contents of register rb. The values are partitioned into groups of operands of the size specified and are multiplied and summed, producing a group of values. The group of values is rounded, and limited as specified, yielding a group of results which is the size specified. The group of results is catenated and placed in register ra.

In an exemplary embodiment, the size of this operation is determined from the contents of register rb. The multiplier usage is constant, but the memory operand size is inversely related to the group size. Presumably this can be checked for cache validity.

In an exemplary embodiment, low order bits of re are used to designate a size, which must be consistent with the group size. Because the memory operand is cached, the size can also be cached, thus eliminating the time required to decode the size, whether from rb or from rc.

In an exemplary embodiment, the wide multiply matrix extract instructions (W.MUL.MAT.X.B, W.MUL.MAT.X.L) perform a partitioned array multiply of up to 16384 bits, that is 128x128 bits. The width of the array can be limited to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one half the desired size in bytes to the virtual address operand: 8, 4, 2, or 1. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one half the desired memory operand size in bytes to the virtual address operand.

As shown in FIG. 15B, in an exemplary embodiment, bits 31 . . . 0 of the contents of register rb specifies several parameters which control the manner in which data is extracted. The position and default values of the control fields allow for the source position to be added to a fixed control value for dynamic computation, and allow for the lower 16 bits of the control field to be set for some of the simpler extract cases by a single GCOPYI instruction.

In an exemplary embodiment, the table below describes the meaning of each label:

label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	reserved
s	1	signed vs. unsigned
n	1	complex vs. real multiplication
m	1	mixed-sign vs. same-sign multiplication
l	1	saturation vs. truncation
rnd	2	rounding
gssp	9	group size and source position

In an exemplary embodiment, the 9 bit gssp field encodes both the group size, gsize, and source position, spos, according to the formula $gssp = 512 \cdot 4 \cdot gsize + spos$. The group size, gsize, is a power of two in the range 1 . . . 128. The source position, spos, is in the range 0 . . . $(2 \cdot gsize) - 1$.

In an exemplary embodiment, the values in the s, n, m, l, and rnd fields have the following meaning:

18

values	s	n	m	l	rnd
0	unsigned	real	same-sign	truncate	F
1	signed	complex	mixed-sign	saturate	Z
2					N
3					C

In an exemplary embodiment, the virtual address must be aligned, that is, it must be an exact multiple of the operand size expressed in bytes. If the address is not aligned an "access disallowed by virtual address" exception occurs.

In an exemplary embodiment, Z (zero) rounding is not defined for unsigned extract operations, and a ReservedInstruction exception is raised if attempted. F (floor) rounding will properly round unsigned results downward.

As shown in FIG. 15C, an exemplary embodiment of a wide-multiply-matrix-extract-complex-doubles instruction (W.MUL.MAT.X.B or W.MUL.MAT.X.L) multiplies memory [m63 m62 m61 . . . m2 m1 m0] with vector [h g f e d c b a], yielding the products

[am7+bm15+cm23+dm31+em39+fm47+gm55+hm63 . . . am2+bm10+cm18+dm26+em34+fm42+gm50+hm58 am1+bm9+cm17+dm25+em33+fm41+gm49+hm57 am0+bm8+cm16+dm24+em32+fm40+gm48+hm56], rounded and limited as specified.

As shown in FIG. 15D, an exemplary embodiment of a wide-multiply-matrix-extract-complex-doubles instruction (W.MUL.MAT.X with n set in rb) multiplies memory [m31 m30 m29 . . . m2 m1 m0] with vector [h g f e d c b a], yielding the products [am7+bm6+cm15+dm14+em23+fm22+gm31+hm30 . . . am2-bm3+cm10-dm11+em18-fm19+gm26-hm27 am1+bm0+cm9+dm8+em17+fm16+gm25+hm24 am0-bm1+cm8-dm9+em16-fm17+gm24 hm25], rounded and limited as specified.

An exemplary embodiment of the pseudocode 1580 of the Wide Multiply Matrix Extract instruction is shown in FIG. 15E. An exemplary embodiment of the exceptions 1590 of the Wide Multiply Matrix Extract instruction is shown in FIG. 15F.

Wide Multiply Matrix Extract Immediate

An exemplary embodiment of the Wide Multiply Matrix Extract Immediate instruction is shown in FIGS. 16A-16E. In an exemplary embodiment, the Wide Multiply Matrix Extract Immediate instructions perform the same function as above, except that the extraction, operand format and size is controlled by fields in the instruction. This form encodes common forms of the above instruction without the need to initialize a register with the required control information. Controls within the instruction allow specification of signed, mixed signed, unsigned, and complex operands.

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands, and catenate the results together, placing the result in a general register. An exemplary embodiment of the format 1610 of the Wide Multiply Matrix Extract Immediate instruction is shown in FIG. 16A.

An exemplary embodiment of the schematics 1630 and 1660 of the Wide Multiply Matrix Extract Immediate instruction is shown in FIGS. 16B and 16C. In an exemplary embodiment, the contents of register rc is used as a virtual address, and a value of specified size is loaded from

US 6,725,356 B2

19

memory. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified and are multiplied and summed in columns, producing a group of sums. The group of sums is rounded, limited, and extracted as specified, yielding a group of results, each of which is the size specified. The group of results is concatenated and placed in register rd. All results are signed, N (nearest) rounding is used, and all results are limited to maximum representable signed values.

In an exemplary embodiment, the wide-multiply-extract-immediate-matrix instructions (W.MUL.MAT.X.I, W.MUL.MAT.X.I.C) perform a partitioned array multiply of up to 16384 bits, that is 128x128 bits. The width of the array can be limited to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired size in bytes to the virtual address operand: 8, 4, 2, or 1. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one half the desired memory operand size in bytes to the virtual address operand.

In an exemplary embodiment, the virtual address must either be aligned to 2048/gsize bytes (or 1024/gsize for W.MUL.MAT.X.I.C), or must be the sum of an aligned address and one-half of the size of the memory operand in bytes and/or one half of the size of the result in bytes. An aligned address must be an exact multiple of the size expressed in bytes. If the address is not valid an "access disallowed by virtual address" exception occurs.

As shown in FIG. 16B, an exemplary embodiment of a wide-multiply-extract-immediate-matrix-doublets instruction (W.MUL.MAT.X.I.16) multiplies memory [m63 m62 m61 ... m2 m1 m0] with vector [h g f e d c b a], yielding the products

[am7+bm15+cm23+dm31+em39+fm47+gm55+hm63 ... am2+bm10+cm18+dm26+em34+fm42+gm50+hm58 am1+bm9+cm17+dm25+em33+fm41 +gm49+hm57 am0+bm8+cm16+dm24+em32+fm40+gm48+hm56], rounded and limited as specified.

As shown in FIG. 16C, an exemplary embodiment of a wide-multiply-matrix-extract-immediate-complex-doublets instruction (W.MUL.MAT.X.I.C.16) multiplies memory [m31 m30 m29 ... m2 m1 m0] with vector [h g f e d c b a], yielding the products [am7+bm6+cm15+dm14+em23+fm22+gm31+hm30 ... am2-bm3+cm10-dm11+em18-fm19+gm26-hm27 am1+bm0+cm9+dm8+em17+fm16+gm25+hm24 am0-bm1+cm8-dm9+em16-fm17+gm24-hm25], rounded and limited as specified.

An exemplary embodiment of the pseudocode 1680 of the Wide Multiply Matrix Extract Immediate instruction is shown in FIG. 16D. An exemplary embodiment of the exceptions 1590 of the Wide Multiply Matrix Extract Immediate instruction is shown in FIG. 16E.

Wide Multiply Matrix Floating-point

An exemplary embodiment of the Wide Multiply Matrix Floating-point instruction is shown in FIGS. 17A-17E. In an exemplary embodiment, the Wide Multiply Matrix Floating-point instructions perform a matrix multiply in the same form as above, except that the multiplies and additions are performed in floating-point arithmetic. Sizes of half (16-bit), single (32-bit), double (64-bit), and complex sizes of half, single and double can be specified within the instruction.

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, a second operand from a general register, perform a group of operations on partitions of bits in the operands,

20

and concatenate the results together, placing the result in a general register. An exemplary embodiment of the format 1710 of the Wide Multiply Matrix Floating-point instruction is shown in FIG. 17A.

An exemplary embodiment of the schematics 1730 and 1760 of the Wide Multiply Matrix Floating-point instruction is shown in FIGS. 17B and 17C. In an exemplary embodiment, the contents of register rc is used as a virtual address, and a value of specified size is loaded from memory. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified. The second values are multiplied with the first values, then summed, producing a group of result values. The group of result values is concatenated and placed in register rd.

In an exemplary embodiment, the wide-multiply-matrix-floating-point instructions (W.MUL.MAT.F, W.MUL.MAT.C.F) perform a partitioned array multiply of up to 16384 bits, that is 128x128 bits. The width of the array can be limited to 128, 64, 32 bits, but not smaller than twice the group size, by adding one-half the desired size in bytes to the virtual address operand: 8, 4, or 2. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size, by adding one-half the desired memory operand size in bytes to the virtual address operand.

In an exemplary embodiment, the virtual address must either be aligned to 2048/gsize bytes (or 1024/gsize for W.MUL.MAT.C.F), or must be the sum of an aligned address and one half of the size of the memory operand in bytes and/or one-half of the size of the result in bytes. An aligned address must be an exact multiple of the size expressed in bytes. If the address is not valid an "access disallowed by virtual address" exception occurs.

As shown in FIG. 17B, an exemplary embodiment of a wide-multiply-matrix-floating-point-half instruction (W.MUL.MAT.F) multiplies memory [m31 m30 ... m1 m0] with vector [h g f e d c b a], yielding products [hm31+gm27+ ... +bm7+am3 ... hm28+gm24+ ... +bm4+am0].

As shown in FIG. 17C, an exemplary embodiment of a wide-multiply-matrix-complex-floating-point-half instruction (W.MUL.MAT.F) multiplies memory [m15 m14 ... m1 m0] with vector [h g f e d c b a], yielding products [hm14+gm15+ ... +bm2+am3 ... hm12+gm13+ ... +bm0+am1-hm13+gm12+ ... -bm1+am0].

An exemplary embodiment of the pseudocode 1780 of the Wide Multiply Matrix Floating-point instruction is shown in FIG. 17D. Additional pseudocode functions used by this and other floating point instructions is shown in FIG. FI.OAT-1.

An exemplary embodiment of the exceptions 1790 of the Wide Multiply Matrix Floating-point instruction is shown in FIG. 17E.

Wide Multiply Matrix Galois

An exemplary embodiment of the Wide Multiply Matrix Galois instruction is shown in FIGS. 18A-18D. In an exemplary embodiment, the Wide Multiply Matrix Galois instructions perform a matrix multiply in the same form as above, except that the multiplies and additions are performed in Galois field arithmetic. A size of 8 bits can be specified within the instruction. The contents of a general register specify the polynomial with which to perform the Galois field remainder operation. The nature of the matrix multiplication is novel and described in detail below.

In an exemplary embodiment, these instructions take an address from a general register to fetch a large operand from memory, second and third operands from general registers,

US 6,725,356 B2

21

perform a group of operations on partitions of bits in the operands, and concatenate the results together, placing the result in a general register. An exemplary embodiment of the format 1810 of the Wide Multiply Matrix Galois instruction is shown in FIG. 18A.

An exemplary embodiment of the schematic 1830 of the Wide Multiply Matrix Galois instruction is shown in FIG. 18B. In an exemplary embodiment, the contents of register *re* is used as a virtual address, and a value of specified size is loaded from memory. Second and third values are the contents of registers *rd* and *rb*. The values are partitioned into groups of operands of the size specified. The second values are multiplied as polynomials with the first value, producing a result which is reduced to the Galois field specified by the third value, producing a group of result values. The group of result values is concatenated and placed in register *ra*.

In an exemplary embodiment, the wide-multiply-matrix-Galois-bytes instruction (W.MUL.MAT.G.8) performs a partitioned array multiply of up to 16384 bits, that is 128x128 bits. The width of the array can be limited to 128, 64, 32, or 16 bits, but not smaller than twice the group size of 8 bits, by adding one-half the desired size in bytes to the virtual address operand: 8, 4, 2, or 1. The array can be limited vertically to 128, 64, 32, or 16 bits, but not smaller than twice the group size of 8 bits, by adding one-half the desired memory operand size, in bytes to the virtual address operand.

In an exemplary embodiment, the virtual address must either be aligned to 256 bytes, or must be the sum of an aligned address and one-half of the size of the memory operand in bytes and/or one-half of the size of the result in bytes. An aligned address must be an exact multiple of the size expressed in bytes. If the address is not valid an "access disallowed by virtual address" exception occurs.

As shown in FIG. 18B, an exemplary embodiment of a wide-multiply-matrix-Galois-byte instruction (W.MUL.MAT.G.8) multiplies memory [m255 m254 ... m1 m0] with vector [p o n m l k j i h g f e d c b a], reducing the result modulo polynomial [q], yielding products [(pm255+om247+...+bm31+am15 mod q) (pm254+om246+...+bm30+am14 mod q) ... (pm248+om240+...+bm16+am0 mod q)].

An exemplary embodiment of the pseudocode 1860 of the Wide Multiply Matrix Galois instruction is shown in FIG. 18C. An exemplary embodiment of the exceptions 1890 of the Wide Multiply Matrix Galois instruction is shown in FIG. 18D.

Memory Operands of Either Little-endian or Big-endian Conventional Byte Ordering

In another aspect of the invention, memory operands of either little-endian or big-endian conventional byte ordering are facilitated. Consequently, all Wide operand instructions are specified in two forms, one for little-endian byte ordering and one for big-endian byte ordering, as specified by a portion of the instruction. The byte order specifies to the memory system the order in which to deliver the bytes within units of the data path width (128 bits), as well as the order to place multiple memory words (128 bits) within a larger Wide operand.

Extraction of a High Order Portion of a Multiplier Product or Sum of Products

Another aspect of the present invention addresses extraction of a high order portion of a multiplier product or sum

22

of products, as a way of efficiently utilizing a large multiplier array. Related U.S. Pat. No. 5,742,840 and U.S. Pat. No. 5,953,241 describe a system and method for enhancing the utilization of a multiplier array by adding specific classes of instructions to a general-purpose processor. This addresses the problem of making the most use of a large multiplier array that is fully used for high-precision arithmetic—for example a 64x64 bit multiplier is fully used by a 64-bit by 64-bit multiply, but only one quarter used for a 32-bit by 32-bit multiply) for (relative to the multiplier data width and registers) low-precision arithmetic operations. In particular, operations that perform a great many low-precision multiplies which are combined (added) together in various ways are specified. One of the overriding considerations in selecting the set of operations is a limitation on the size of the result operand. In an exemplary embodiment, for example, this size might be limited to on the order of 128 bits, or a single register, although no specific size limitation need exist.

The size of a multiply result, a product, is generally the sum of the sizes of the operands, multiplicands and multiplier. Consequently, multiply instructions specify operations in which the size of the result is twice the size of identically-sized input operands. For our prior art design, for example, a multiply instruction accepted two 64-bit register sources and produces a single 128-bit register-pair result, using an entire 64x64 multiplier array for 64-bit symbols, or half the multiplier array for pairs of 32-bit symbols, or one quarter the multiplier array for quads of 16-bit symbols. For all of these cases, note that two register sources of 64 bits are combined, yielding a 128-bit result.

In several of the operations, including complex multiplies, convolve, and matrix multiplication, low-precision multiplier products are added together. The additions further increase the required precision. The sum of two products requires one additional bit of precision; adding four products requires two, adding eight products requires three, adding sixteen products requires four. In some prior designs, some of this precision is lost, requiring scaling of the multiplier operands to avoid overflow, further reducing accuracy of the result.

The use of register pairs creates an undesirable complexity, in that both the register pair and individual register values must be bypassed to subsequent instructions. As a result, with prior art techniques only half of the source operand 128-bit register values could be employed toward producing a single-register 128-bit result.

In the present invention, a high-order portion of the multiplier product or sum of products is extracted, adjusted by a dynamic shift amount from a general register or an adjustment specified as part of the instruction, and rounded by a control value from a register or instruction portion as round-to-nearest/even, toward zero, floor, or ceiling. Overflows are handled by limiting the result to the largest and smallest values that can be accurately represented in the output result.

Extract Controlled by a Register

In the present invention, when the extract is controlled by a register, the size of the result can be specified, allowing rounding and limiting to a smaller number of bits than can fit in the result. This permits the result to be scaled to be used in subsequent operations without concern of overflow or rounding, enhancing performance.

Also in the present invention, when the extract is controlled by a register, a single register value defines the size

US 6,725,356 B2

23

of the operands, the shift amount and size of the result, and the rounding control. By placing all this control information in a single register, the size of the instruction is reduced over the number of bits that such a instruction would otherwise require, improving performance and enhancing flexibility of the processor.

The particular instructions included in this aspect of the present invention are Ensemble Convolve Extract, Ensemble Multiply Extract, Ensemble Multiply Add Extract and Ensemble Scale Add Extract.

Ensemble Extract Inplace

An exemplary embodiment of the Ensemble Extract Inplace instruction is shown in FIGS. 19A–19G. In an exemplary embodiment, several of these instructions (Ensemble Convolve Extract, Ensemble Multiply Add Extract) are typically available only in forms where the extract is specified as part of the instruction. An alternative embodiment can incorporate forms of the operations in which the size of the operand, the shift amount and the rounding can be controlled by the contents of a general register (as they are in the Ensemble Multiply Extract instruction). The definition of this kind of instruction for Ensemble Convolve Extract, and Ensemble Multiply Add Extract would require four source registers, which increases complexity by requiring additional general-register read ports.

In an exemplary embodiment, these operations take operands from four registers, perform operations on partitions of bits in the operands, and place the concatenated results in a fourth register. An exemplary embodiment of the format and operation codes 1910 of the Ensemble Extract Inplace instruction is shown in FIG. 19A.

An exemplary embodiment of the schematics 1930, 1945, 1960, and 1975 of the Ensemble Extract Inplace instruction is shown in FIGS. 19C, 19D, 19E, and 19F. In an exemplary embodiment, the contents of registers rd, rc, rb, and ra are fetched. The specified operation is performed on these operands. The result is placed into register rd.

In an exemplary embodiment, for the E.CON.X instruction, the contents of registers rd and rc are concatenated, as c d, and used as a first value. A second value is the contents of register rb. The values are partitioned into groups of operands of the size specified and are convolved, producing a group of values. The group of values is rounded, limited and extracted as specified, yielding a group of results that is the size specified. The group of results is concatenated and placed in register rd.

In an exemplary embodiment, for the E.MUL.ADD.X instruction, the contents of registers rc and rb are partitioned into groups of operands of the size specified and are multiplied, producing a group of values to which are added the partitioned and extended contents of register rd. The group of values is rounded, limited and extracted as specified, yielding a group of results that is the size specified. The group of results is concatenated and placed in register rd.

As shown in FIG. 19B, in an exemplary embodiment, bits 31 . . . 0 of the contents of register ra specifies several parameters that control the manner in which data is extracted, and for certain operations, the manner in which the operation is performed. The position of the control fields allows for the source position to be added to a fixed control value for dynamic computation, and allows for the lower 16 bits of the control field to be set for some of the simpler extract cases by a single GCOPY128 instruction. The

24

control fields are further arranged so that if only the low order 8 bits are non-zero, a 128-bit extraction with truncation and no rounding is performed.

In an exemplary embodiment, the table below describes the meaning of each label:

label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	extended vs. group size result
s	1	signed vs. unsigned
n	1	complex vs. real multiplication
m	1	mixed-sign vs. same-sign multiplication
l	1	limit: saturation vs. truncation
rnd	2	rounding
gssp	9	group size and source position

In an exemplary embodiment, the 9-bit gssp field encodes both the group size, gsize, and source position, spos, according to the formula $gssp = 512 - 4 * gsize + spos$. The group size, gsize, is a power of two in the range 1 . . . 128. The source position, spos, is in the range 0 . . . $(2 * gsize) - 1$.

In an exemplary embodiment, the values in the x, s, n, m, l, and rnd fields have the following meaning:

values	x	s	n	m	l	rnd
0	group	unsigned	real	same-sign	truncate	F
1	extended	signed	complex	mixed-sign	saturation	Z
2						N
3						C

Ensemble Multiply Add Extract

As shown in FIG. 19C, an exemplary embodiment of an ensemble-multiply-add-extract-doublets instruction (E.MUL.ADDX) multiplies vector rc [h g f e d c b a] with vector rb [p o n m l k j i], and adding vector rd [x w v u t s r q], yielding the result vector rd [hp+x go+w fn+v em+u dl+t ck+s bj+r ai+q], rounded and limited as specified by ra31 . . . 0.

As shown in FIG. 19D, an exemplary embodiment of an ensemble-multiply-add-extract-doublets-complex instruction (E.MUL.X with n set) multiplies operand vector rc [h g f e d c b a] by operand vector rb [p o n m l k j i], yielding the result vector rd [gp+ho go-hp en+fm em-fn el+dk ek-dl aj+bi ai-bj], rounded and limited as specified by ra31 . . . 0. Note that this instruction prefers an organization of complex numbers in which the real part is located to the right (lower precision) of the imaginary part.

Ensemble Convolve Extract

As shown in FIG. 19E, an exemplary embodiment of an ensemble-convolve-extract-doublets instruction (ECON.X with n=0) convolves vector rc r d [x w v u t s r q p o n m l k j i] with vector rb [h g f e d c b a], yielding the products vector rd

[ax+bw+cv+du+et+fs+gr+hq . . . as+br+cq+dp+eo+fn+gm+hl ar+bq+cp+do+en+fm+gl+hk aq+bp+co+dn+em+fl+gk+hj], rounded and limited as specified by ra31 . . . 0.

As shown in FIG. 19F, an exemplary embodiment of an ensemble-convolve-extract-complex-doublets instruction

US 6,725,356 B2

25

(ECON.X with n=1) convolves vector rd rc [x w v u t s r q p o n m l k j i] with vector rb [h g f e d c b a], yielding the products vector rd

[ax+bw+cv+du+et+fs+gr+hq . . . as-bt+cq-dr+eo-fp+gm-hn ar+bq+cp+do+en+fm+gl+hk aq-br+co-dp+em-fn+gk+hl], rounded and limited as specified by ra31 . . . 0.

An exemplary embodiment of the pseudocode 1990 of Ensemble Extract Inplace instruction is shown in FIG. 19G. In an exemplary embodiment, there are no exceptions for the Ensemble Extract Inplace instruction.

Ensemble Extract

An exemplary embodiment of the Ensemble Extract instruction is shown in FIGS. 20A–20J. In an exemplary embodiment, these operations take operands from three registers, perform operations on partitions of bits in the operands, and place the concatenated results in a fourth register. An exemplary embodiment of the format and operation codes 2010 of the Ensemble Extract instruction is shown in FIG. 20A.

An exemplary embodiment of the schematics 2020, 2030, 2040, 2050, 2060, 2070, and 2080 of the Ensemble Extract Inplace instruction is shown in FIGS. 20C, 20D, 20E, 20F, 20G, 20H, and 20I. In an exemplary embodiment, the contents of registers rd, rc, and rb are fetched. The specified operation is performed on these operands. The result is placed into register ra.

As shown in FIG. 20B, in an exemplary embodiment, bits 31 . . . 0 of the contents of register rb specifies several parameters that control the manner in which data is extracted, and for certain operations, the manner in which the operation is performed. The position of the control fields allows for the source position to be added to a fixed control value for dynamic computation, and allows for the lower 16 bits of the control field to be set for some of the simpler extract cases by a single GCOPYL128 instruction. The control fields are further arranged so that if only the lower order 8 bits are non-zero, a 128-bit extraction with truncation and no rounding is performed.

In an exemplary embodiment, the table below describes the meaning of each label:

label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	extended vs. group size result
s	1	signed vs. unsigned
n	1	complex vs. real multiplication
m	1	merge vs. extract or mixed-sign vs. same-sign multiplication
l	1	limit: saturation vs. truncation
rad	2	rounding
gssp	9	group size and source position

In an exemplary embodiment, the 9-bit gssp field encodes both the group size, gsize, and source position, spos, according to the formula $gssp = 512 \cdot 4 \cdot gsize + spos$. The group size, gsize, is a power of two in the range 1 . . . 128. The source position, spos, is in the range 0 . . . $(2 \cdot gsize) - 1$.

In an exemplary embodiment, the values in the x, s, n, m, l, and rad fields have the following meaning:

26

values	x	s	n	m	l	rad
0	group	unsigned	real	extract/ same-sign	truncate	F
1	extended	signed	complex	merge/ mixed-sign	saturation	Z
2						N
3						C

In an exemplary embodiment, for the E.SCAL.ADD.X instruction, bits 127 . . . 64 of the contents of register rb specifies the multipliers for the multiplicands in registers rd and rc. Specifically, bits $64 + 2 \cdot gsize - 1$. . . $64 + gsize$ is the multiplier for the contents of register rd, and bits $64 + gsize - 1$. . . 64 is the multiplier for the contents of register rc.

Ensemble Multiply Extract

As shown in FIG. 20C, an exemplary embodiment of an ensemble-multiply-extract-doubles instruction (E.MULX) multiplies vector rd [h g f e d c b a] with vector rc [p o n m l k j i], yielding the result vector ra [hp go fn em dl ck bj ai], rounded and limited as specified by rb₃₁ . . . 0.

As shown in FIG. 20D, an exemplary embodiment of an ensemble-multiply-extract-doubles-complex instruction (E.MULX with n set) multiplies vector rd [h g f e d c b a] by vector rc [p o n m l k j i], yielding the result vector ra [gp+ho go-hp en+fm em-fn cl+dk ck-dl aj+bi ai-bj], rounded and limited as specified by rb₃₁ . . . 0. Note that this instruction prefers an organization of complex numbers in which the real part is located to the right (lower precision) of the imaginary part.

Ensemble Scale Add Extract

An aspect of the present invention defines the Ensemble Scale Add Extract instruction, that combines the extract control information in a register along with two values that are used as scalar multipliers to the contents of two vector multiplicands.

This combination reduces the number of registers that would otherwise be required, or the number of bits that the instruction would otherwise require, improving performance. Another advantage of the present invention is that the combined operation may be performed by an exemplary embodiment with sufficient internal precision on the summation node that no intermediate rounding or overflow occurs, improving the accuracy over prior art operation in which more than one instruction is required to perform this computation.

As shown in FIG. 20E, an exemplary embodiment of an ensemble-scale-add-extract-doubles instruction (E.SCAL.ADD.X) multiplies vector rd [h g f e d c b a] with rb₉₅ . . . 80 [r] and adds the product to the product of vector rc [p o n m l k j i] with rb₇₉ . . . 64 [q], yielding the result [hr+pq gr+oq fr+nq er+mq dr+lq cr+kq br+jq ar+iq], rounded and limited as specified by rb₃₁ . . . 0.

As shown in FIG. 20F, an exemplary embodiment of an ensemble-scale-add-extract-doubles-complex instruction (E.SCALADD.X with n set) multiplies vector rd [h g f e d c b a] with rb₁₂₇ . . . 96 [r s] and adds the product to the product of vector rc [p o n m l k j i] with rb₉₅ . . . 64 [r q], yielding the result [hs+gt+pq+or gs-ht+oq-pr fs+et+nq+mr es-ft+mq-nr ds+ct+lq+kr cs-dt+kq-lr bs+at+jq+ir as-bt+iq-jr], rounded and limited as specified by rb₃₁ . . . 0.

Ensemble Extract

As shown in FIG. 20G, in an exemplary embodiment, for the E.EXTRACT instruction, when m=0 and x=0, the

US 6,725,356 B2

27

parameters specified by the contents of register rb are interpreted to select fields from double size symbols of the catenated contents of registers rd and rc, extracting values which are catenated and placed in register ra.

As shown in FIG. 20H, in an exemplary embodiment, for an ensemble-merge-extract (E.EXTRACT when $m=1$), the parameters specified by the contents of register rb are interpreted to merge fields from symbols of the contents of register rd with the contents of register rc. The results are catenated and placed in register ra. The x field has no effect when $m=1$.

As shown in FIG. 20I, in an exemplary embodiment, for an ensemble-expand-extract (E.EXTRACT when $m=0$ and $x=1$), the parameters specified by the contents of register rb are interpreted to extract fields from symbols of the contents of register rd. The results are catenated and placed in register ra. Note that the value of rc is not used.

An exemplary embodiment of the pseudocode 2090 of Ensemble Extract instruction is shown in FIG. 20J. In an exemplary embodiment, there are no exceptions for the Ensemble Extract instruction.

Reduction of Register Read Ports

Another alternative embodiment can reduce the number of register read ports required for implementation of instructions in which the size, shift and rounding of operands is controlled by a register. The value of the extract control register can be fetched using an additional cycle on an initial execution and retained within or near the functional unit for subsequent executions, thus reducing the amount of hardware required for implementation with a small additional performance penalty. The value retained would be marked invalid, causing a re-fetch of the extract control register, by instructions that modify the register, or alternatively, the retained value can be updated by such an operation. A re-fetch of the extract control register would also be required if a different register number were specified on a subsequent execution. It should be clear that the properties of the above two alternative embodiments can be combined.

Galois Field Arithmetic

Another aspect of the invention includes Galois field arithmetic, where multiplies are performed by an initial binary polynomial multiplication (unsigned binary multiplication with carries suppressed), followed by a polynomial modulo/remainder operation (unsigned binary division with carries suppressed). The remainder operation is relatively expensive in area and delay. In Galois field arithmetic, additions are performed by binary addition with carries suppressed, or equivalently, a bitwise exclusive or operation. In this aspect of the present invention, a matrix multiplication is performed using Galois field arithmetic, where the multiplies and additions are Galois field multiples and additions.

Using prior art methods, a 16 byte vector multiplied by a 16×16 byte matrix can be performed as 256 8-bit Galois field multiplies and $16 \times 15 = 240$ 8-bit Galois field additions. Included in the 256 Galois field multiplies are 256 polynomial multiplies and 256 polynomial remainder operations.

By use of the present invention, the total computation is reduced significantly by performing 256 polynomial multiplies, 240 16-bit polynomial additions, and 16 polynomial remainder operations. Note that the cost of the polynomial additions has been doubled compared with the Galois field additions, as these are now 16-bit operations

28

rather than 8-bit operations, but the cost of the polynomial remainder functions has been reduced by a factor of 16. Overall, this is a favorable tradeoff, as the cost of addition is much lower than the cost of remainder.

Decoupled Access From Execution Pipelines and Simultaneous Multithreading

In yet another aspect of the present invention, best shown in FIG. 4, the present invention employs both decoupled access from execution pipelines and simultaneous multithreading in a unique way. Simultaneous Multithreaded pipelines have been employed in prior art to enhance the utilization of data path units by allowing instructions to be issued from one of several execution threads to each functional unit (e.g. Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy, "Simultaneous Multithreading: Maximizing On Chip Parallelism," Proceedings of the 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June, 1995).

Decoupled access from execution pipelines have been employed in prior art to enhance the utilization of execution data path units by buffering results from an access unit, which computes addresses to a memory unit that in turn fetches the requested items from memory, and then presenting them to an execution unit (e.g. J. E. Smith, "Decoupled Access/Execute Computer Architectures", Proceedings of the Ninth Annual International Symposium on Computer Architecture, Austin, Tex. (Apr. 26-29, 1982), pp. 112-119).

Compared to conventional pipelines, the Eggers prior art used an additional pipeline cycle before instructions could be issued to functional units, the additional cycle needed to determine which threads should be permitted to issue instructions. Consequently, relative to conventional pipelines, the prior art design had additional delay, including dependent branch delay.

The present invention contains individual access data path units, with associated register files, for each execution thread. These access units produce addresses, which are aggregated together to a common memory unit, which fetches all the addresses and places the memory contents in one or more buffers. Instructions for execution units, which are shared to varying degrees among the threads are also buffered for later execution. The execution units then perform operations from all active threads using functional data path units that are shared.

For instructions performed by the execution units, the extra cycle required for prior art simultaneous multithreading designs is overlapped with the memory data access time from prior art decoupled access from execution cycles, so that no additional delay is incurred by the execution functional units for scheduling resources. For instructions performed by the access units, by employing individual access units for each thread the additional cycle for scheduling shared resources is also eliminated.

This is a favorable tradeoff because, while threads do not share the access functional units, these units are relatively small compared to the execution functional units, which are shared by threads.

With regard to the sharing of execution units, the present invention employs several different classes of functional units for the execution unit, with varying cost, utilization, and performance. In particular, the G units, which perform simple addition and bitwise operations is relatively inexpensive (in area and power) compared to the other units, and its utilization is relatively high. Consequently, the design employs four such units, where each unit can be shared

US 6,725,356 B2

29

between two threads. The X unit, which performs a broad class of data switching functions is more expensive and less used, so two units are provided that are each shared among two threads. The T unit, which performs the Wide Translate instruction, is expensive and utilization is low, so the single unit is shared among all four threads. The E unit, which performs the class of Ensemble instructions, is very expensive in area and power compared to the other functional units, but utilization is relatively high, so we provide two such units, each unit shared by two threads.

In FIG. 4, four copies of an access unit are shown, each with an access instruction fetch queue A-Queue 401-404, coupled to an access register file AR 405-408, each of which is, in turn, coupled to two access functional units A 409-416. The access units function independently for four simultaneous threads of execution. These eight access functional units A 409-416 produce results for access register files AR 405-408 and addresses to a shared memory system 417. The memory contents fetched from memory system 417 are combined with execute instructions not performed by the access unit and entered into the four execute instruction queues E-Queue 421-424. Instructions and memory data from E-queue 421-424 are presented to execution register files 425-428, which fetches execution register file source operands. The instructions are coupled to the execution unit arbitration unit Arbitration 431, that selects which instructions from the four threads are to be routed to the available execution units E 441 and 449, X 442 and 448, G 443-444 and 446-447, and T 445. The execution register file source operands ER 425-428 are coupled to the execution units 441-445 using source operand buses 451-454 and to the execution units 445-449 using source operand buses 455-458. The function unit result operands from execution units 441-445 are coupled to the execution register file using result bus 461 and the function units result operands from execution units 445-449 are coupled to the execution register file using result bus 462.

Improved Interprivilege Gateway

In a still further aspect of the present invention, an improved interprivilege gateway is described which involves increased parallelism and leads to enhanced performance. In related U.S. patent application Ser. No. 08/541, 416, a system and method is described for implementing an instruction that, in a controlled fashion, allows the transfer of control (branch) from a lower privilege level to a higher privilege level. The present invention is an improved system and method for a modified instruction that accomplishes the same purpose but with specific advantages.

Many processor resources, such as control of the virtual memory system itself, input and output operations, and system control functions are protected from accidental or malicious misuse by enclosing them in a protective, privileged region. Entry to this region must be established only through particular entry points, called gateways, to maintain the integrity of these protected regions.

Prior art versions of this operation generally load an address from a region of memory using a protected virtual memory attribute that is only set for data regions that contain valid gateway entry points, then perform a branch to an address contained in the contents of memory. Basically, three steps were involved: load, then branch and check. Compared to other instructions, such as register to register computation instructions and memory loads and stores, and register based branches, this is a substantially longer operation, which introduces delays and complexity to a pipelined implementation.

30

In the present invention, the branch-gateway instruction performs two operations in parallel: 1) a branch is performed to the Contents of register 0 and 2) a load is performed using the contents of register 1, using a specified byte order (little-endian) and a specified size (64 bits). If the value loaded from memory does not equal the contents of register 0, the instruction is aborted due to an exception. In addition, 3) a return address (the next sequential instruction address following the branch-gateway instruction) is written into register 0, provided the instruction is not aborted. This approach essentially uses a first instruction to establish the requisite permission to allow user code to access privileged code, and then a second instruction is permitted to branch directly to the privileged code because of the permissions issued for the first instruction.

In the present invention, the new privilege level is also contained in register 0, and the second parallel operation does not need to be performed if the new privilege level is not greater than the old privilege level. When this second operation is suppressed, the remainder of the instruction performs an identical function to a branch-link instruction, which is used for invoking procedures that do not require an increase in privilege. The advantage that this feature brings is that the branch-gateway instruction can be used to call a procedure that may or may not require an increase in privilege.

The memory load operation verifies with the virtual memory system that the region that is loaded has been tagged as containing valid gateway data. A further advantage of the present invention is that the called procedure may rely on the fact that register 1 contains the address that the gateway data was loaded from, and can use the contents of register 1 to locate additional data or addresses that the procedure may require. Prior art versions of this instruction required that an additional address be loaded from the gateway region of memory in order to initialize that address in a protected manner—the present invention allows the address itself to be loaded with a “normal” load operation that does not require special protection.

The present invention allows a “normal” load operation to also load the contents of register 0 prior to issuing the branch-gateway instruction. The value may be loaded from the same memory address that is loaded by the branch-gateway instruction, because the present invention contains a virtual memory system in which the region may be enabled for normal load operations as well as the special “gateway” load operation performed by the branch-gateway instruction.

Improved Interprivilege Gateway—System and Privileged Library Calls

An exemplary embodiment of the System and Privileged Library Calls is shown in FIGS. 21A-21B. An exemplary embodiment of the schematic 2110 of System and Privileged Library Calls is shown in FIG. 21A. In an exemplary embodiment, it is an objective to make calls to system facilities and privileged libraries as similar as possible to normal procedure calls as described above. Rather than invoke system calls as an exception, which involves significant latency and complication, a modified procedure call in which the process privilege level is quietly raised to the required level is used. To provide this mechanism safely, interaction with the virtual memory system is required.

In an exemplary embodiment, such a procedure must not be entered from anywhere other than its legitimate entry point, to prohibit entering a procedure after the point at which security checks are performed or with invalid register

US 6,725,356 B2

31

contents, otherwise the access to a higher privilege level can lead to a security violation. In addition, the procedure generally must have access to memory data, for which addresses must be produced by the privileged code. To facilitate generating these addresses, the branch-gateway instruction allows the privileged code procedure to rely on the fact that a single register has been verified to contain a pointer to a valid memory region.

In an exemplary embodiment, the branch-gateway instruction ensures both that the procedure is invoked at a proper entry point, and that other registers such as the data pointer and stack pointer can be properly set. To ensure this, the branch-gateway instruction retrieves a "gateway" directly from the protected virtual memory space. The gateway contains the virtual address of the entry point of the procedure and the target privilege level. A gateway can only exist in regions of the virtual address space designated to contain them, and can only be used to access privilege levels at or below the privilege level at which the memory region can be written to ensure that a gateway cannot be forged.

In an exemplary embodiment, the branch-gateway instruction ensures that register 1 (dp) contains a valid pointer to the gateway for this target code address by comparing the contents of register 0 (lp) against the gateway retrieved from memory and causing an exception trap if they do not match. By ensuring that register 1 points to the gateway, auxiliary information, such as the data pointer and stack pointer can be set by loading values located by the contents of register 1. For example, the eight bytes following the gateway may be used as a pointer to a data region for the procedure.

In an exemplary embodiment, before executing the branch-gateway instruction, register 1 must be set to point at the gateway, and register 0 must be set to the address of the target code address plus the desired privilege level. A "L.L.64.L.A r0=r1,0" instruction is one way to set register 0, if register 1 has already been set, but any means of getting the correct value into register 0 is permissible.

In an exemplary embodiment, similarly, a return from a system or privileged routine involves a reduction of privilege. This need not be carefully controlled by architectural facilities, so a procedure may freely branch to a less-privileged code address. Normally, such a procedure restores the stack frame, then uses the branch-down instruction to return.

An exemplary embodiment of the typical dynamic-linked, inter-gateway calling sequence 2130 is shown in FIG. 21B. In an exemplary embodiment, the calling sequence is identical to that of the inter-module calling sequence shown above, except for the use of the B.GATE instruction instead of a B.LINK instruction. Indeed, if a B.GATE instruction is used when the privilege level in the lp register is not higher than the current privilege level, the B.GATE instruction performs an identical function to a B.LINK.

In an exemplary embodiment, the callee, if it uses a stack for local variable allocation, cannot necessarily trust the value of the sp passed to it, as it can be forged. Similarly, any pointers which the callee provides should not be used directly unless it they are verified to point to regions which the callee should be permitted to address. This can be avoided by defining application programming interfaces (APIs) in which all values are passed and returned in registers, or by using a trusted, intermediate privilege wrapper routine to pass and return parameters. The method described below can also be used.

In an exemplary embodiment, it can be useful to have highly privileged code call less-privileged routines. For

32

example, a user may request that errors in a privileged routine be reported by invoking a user-supplied error-logging routine. To invoke the procedure, the privilege can be reduced via the branch-down instruction. The return from the procedure actually requires an increase in privilege, which must be carefully controlled. This is dealt with by placing the procedure call within a lower-privilege procedure wrapper, which uses the branch-gateway instruction to return to the higher privilege region after the call through a secure re-entry point. Special care must be taken to ensure that the less-privileged routine is not permitted to gain unauthorized access by corruption of the stack or saved registers, such as by saving all registers and setting up a new stack frame (or restoring the original lower-privilege stack) that may be manipulated by the less-privileged routine. Finally, such a technique is vulnerable to an unprivileged routine attempting to use the re-entry point directly, so it may be appropriate to keep a privileged state variable which controls permission to enter at the re-entry point.

Improved Interprivilege Gateway—Branch Gateway

An exemplary embodiment of the Branch Gateway instruction is shown in FIGS. 21C–21F. In an exemplary embodiment, this operation provides a secure means to call a procedure, including those at a higher privilege level. An exemplary embodiment of the format and operation codes 2160 of the Branch Gateway instruction is shown in FIG. 21C.

An exemplary embodiment of the schematic 2170 of the Branch Gateway instruction is shown in FIG. 21D. In an exemplary embodiment, the contents of register rb is a branch address in the high-order 62 bits and a new privilege level in the low-order 2 bits. A branch and link occurs to the branch address, and the privilege level is raised to the new privilege level. The high-order 62 bits of the successor to the current program counter is concatenated with the 2-bit current execution privilege and placed in register 0.

In an exemplary embodiment, if the new privilege level is greater than the current privilege level, an octet of memory data is fetched from the address specified by register 1, using the little-endian byte order and a gateway access type. A GatewayDisallowed exception occurs if the original contents of register 0 do not equal the memory data.

In an exemplary embodiment, if the new privilege level is the same as the current privilege level, no checking of register 1 is performed.

In an exemplary embodiment, an AccessDisallowed exception occurs if the new privilege level is greater than the privilege level required to write the memory data, or if the old privilege level is lower than the privilege required to access the memory data as a gateway, or if the access is not aligned on an 8-byte boundary.

In an exemplary embodiment, a ReservedInstruction exception occurs if the rc field is not one or the rd field is not zero.

In an exemplary embodiment, in the example in FIG. 21D, a gateway from level 0 to level 2 is illustrated. The gateway pointer, located by the contents of register rc (1), is fetched from memory and compared against the contents of register rb (0). The instruction may only complete if these values are equal. Concurrently, the contents of register rb (0) is placed in the program counter and privilege level, and the address of the next sequential address and privilege level is placed into register rd (0). Code at the target of the gateway locates the data pointer at an offset from the gateway pointer (register 1), and fetches it into register 1, making a data

US 6,725,356 B2

33

region available. A stack pointer may be saved and fetched using the data region, another region located from the data region, or a data region located as an offset from the original gateway pointer.

In an exemplary embodiment, this instruction gives the target procedure the assurances that register 0 contains a valid return address and privilege level, that register 1 points to the gateway location, and that the gateway location is octlet aligned. Register 1 can then be used to securely reach values in memory. If no sharing of literal pools is desired, register 1 may be used as a literal pool pointer directly. If sharing of literal pools is desired, register 1 may be used with an appropriate offset to load a new literal pool pointer; for example, with a one cache line offset from the register 1. Note that because the virtual memory system operates with cache line granularity, that several gateway locations must be created together.

In an exemplary embodiment, software must ensure that an attempt to use any octlet within the region designated by virtual memory as gateway either functions properly or causes a legitimate exception. For example, if the adjacent octlets contain pointers to literal pool locations, software should ensure that these literal pools are not executable, or that by virtue of being aligned addresses, cannot raise the execution privilege level. If register 1 is used directly as a literal pool location, software must ensure that the literal pool locations that are accessible as a gateway do not lead to a security violation.

In an exemplary embodiment, register 0 contains a valid return address and privilege level, the value is suitable for use directly in the Branch down (B.DOWN) instruction to return to the gateway callee.

An exemplary embodiment of the pseudocode 2190 of the Branch Gateway instruction is shown in FIG. 21E. An exemplary embodiment of the exceptions 2199 of the Branch Gateway instruction is shown in FIG. 21F.

Group Add

In accordance with one embodiment of the invention, the processor handles a variety fix-point, or integer, group operations. For example, FIG. 26A presents various examples of Group Add instructions accommodating different operand sizes, such as a byte (8 bits), doublet (16 bits), quadlet (32 bits), octlet (64 bits), and hexlet (128 bits). FIGS. 26B and 26C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Group Add instructions shown in FIG. 26A. As shown in FIGS. 26B and 26C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified and added, and if specified, checked for overflow or limited, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register rd. While the use of two operand registers and a different result register is described here and elsewhere in the present specification, other arrangements, such as the use of immediate values, may also be implemented.

In the present embodiment, for example, if the operand size specified is a byte (8 bits), and each register is 128-bit wide, then the content of each register may be partitioned into 16 individual operands, and 16 different individual add operations may take place as the result of a single Group Add instruction. Other instructions involving groups of operands may perform group operations in a similar fashion.

Group Set and Group Subtract

Similarly, FIG. 27A presents various examples of Group Set instructions and Group Subtract instructions accommo-

34

dating different operand sizes. FIGS. 27B and 27C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Group Set instructions and Group Subtract instructions. As shown in FIGS. 27B and 27C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified and for Group Set instructions are compared for a specified arithmetic condition or for Group Subtract instructions are subtracted, and if specified, checked for overflow or limited, yielding a group of results, each of which is the size specified. The group of results is catenated and placed in register rd.

Ensemble Convolve, Divide, Multiply, Multiply Sum

In the present embodiment, other fix-point group operations are also available. FIG. 28A presents various examples of Ensemble Convolve, Ensemble Divide, Ensemble Multiply, and Ensemble Multiply Sum instructions accommodating different operand sizes. FIGS. 28B and 28C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Ensemble Convolve, Ensemble Divide, Ensemble Multiply and Ensemble Multiply Sum instructions. As shown in FIGS. 28B and 28C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified and convolved or divided or multiplied, yielding a group of results, or multiplied and summed to a single result. The group of results is catenated and placed, or the single result is placed, in register rd.

Ensemble Floating-point Add, Divide, Multiply, and Subtract

In accordance with one embodiment of the invention, the processor also handles a variety floating-point group operations accommodating different operand sizes. Here, the different operand sizes may represent floating point operands of different precisions, such as half-precision (16 bits), single-precision (32 bits), double-precision (64 bits), and quad-precision (128 bits). FIG. 29 illustrates exemplary functions that are defined for use within the detailed instruction definitions in other sections and figures. In the functions set forth in FIG. 29, an internal format represents infinite-precision floating-point values as a four-element structure consisting of (1) s (sign bit): 0 for positive, 1 for negative, (2) t (type): NORM, ZERO, SNAN, QNAN, INFINITY, (3) e (exponent), and (4) f: (fraction). The mathematical interpretation of a normal value places the binary point at the units of the fraction, adjusted by the exponent: $(-1)^s \cdot 2^e \cdot f$. The function F converts a packed IEEE floating-point value into internal format. The function PackF converts an internal format back into IEEE floating-point format, with rounding and exception control.

FIGS. 30A and 31A present various examples of Ensemble Floating Point Add, Divide, Multiply, and Subtract instructions. FIGS. 30B-C and 31B-C illustrate an exemplary embodiment of formats and operation codes that can be used to perform the various Ensemble Floating Point Add, Divide, Multiply, and Subtract instructions. In these examples, Ensemble Floating Point Add, Divide, and Multiply instructions have been labeled as "EnsembleFloatingPoint." Also, Ensemble Floating-Point Subtract instructions have been labeled as "EnsembleReversedFloatingPoint." As shown in FIGS. 30B-C and 31B-C, in this exemplary embodiment, the contents of registers rc and rb are partitioned into groups of operands of the size specified, and the

US 6,725,356 B2

35

specified group operation is performed, yielding a group of results. The group of results is concatenated and placed in register rd.

In the present embodiment, the operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

Ensemble Scale-Add Floating-point

A novel instruction, Ensemble-Scale-Add improves processor performance by performing two sets of parallel multiplications and pairwise summing the products. This improves performance for operations in which two vectors must be scaled by two independent values and then summed, providing two advantages over nearest prior art operations of a fused-multiply-add. To perform this operation using prior art instructions, two instructions would be needed, an ensemble-multiply for one vector and one scaling value, and an ensemble-multiply-add for the second vector and second scaling value, and these operations are clearly dependent. In contrast, the present invention fuses both the two multiplies and the addition for each corresponding elements of the vectors into a single operation. The first advantage achieved is improved performance, as in an exemplary embodiment the combined operation performs a greater number of multiplies in a single operation, thus improving utilization of the partitioned multiplier unit. The second advantage achieved is improved accuracy, as an exemplary embodiment may compute the fused operation with sufficient intermediate precision so that no intermediate rounding the products is required.

An exemplary embodiment of the Ensemble Scale-Add Floating-point instruction is shown in FIGS. 22A–22B. In an exemplary embodiment, these operations take three values from registers, perform a group of floating-point arithmetic operations on partitions of bits in the operands, and place the concatenated results in a register. An exemplary embodiment of the format 2210 of the Ensemble Scale-Add Floating-point instruction is shown in FIG. 22A.

In an exemplary embodiment, the contents of registers rd and rc are taken to represent a group of floating-point operands. Operands from register rd are multiplied with a floating-point operand taken from the least-significant bits of the contents of register rb and added to operands from register rc multiplied with a floating-point operand taken from the next least-significant bits of the contents of register rb. The results are rounded to the nearest representable floating-point value in a single floating-point operation. Floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754. The results are concatenated and placed in register ra.

An exemplary embodiment of the pseudocode 2230 of the Ensemble Scale-Add Floating-point instruction is shown in FIG. 22B. In an exemplary embodiment, there are no exceptions for the Ensemble Scale-Add Floating-point instruction.

Performing a Three-input Bitwise Boolean Operation in a Single Instruction (Group Boolean)

In a further aspect of the present invention, a system and method is provided for performing a three-input bitwise Boolean operation in a single instruction. A novel method is

36

used to encode the eight possible output states of such an operation into only seven bits, and decoding these seven bits back into the eight states.

An exemplary embodiment of the Group Boolean instruction is shown in FIGS. 23A–23C. In an exemplary embodiment, these operations take operands from three registers, perform boolean operations on corresponding bits in the operands, and place the concatenated results in the third register. An exemplary embodiment of the format 2310 of the Group Boolean instruction is shown in FIG. 23A.

An exemplary embodiment of a procedure 2320 of Group Boolean instruction is shown in FIG. 23B. In an exemplary embodiment, three values are taken from the contents of registers rd, rc and rb. The ih and il fields specify a function of three bits, producing a single bit result. The specified function is evaluated for each bit position, and the results are concatenated and placed in register rd. In an exemplary embodiment, register rd is both a source and destination of this instruction.

In an exemplary embodiment, the function is specified by eight bits, which give the result for each possible value of the three source bits in each bit position:

d	1 1 1 1 0 0 0 0
c	1 1 0 0 1 1 0 0
b	1 0 1 0 1 0 1 0
f(d,c,b)	f ₇ f ₆ f ₅ f ₄ f ₃ f ₂ f ₁ f ₀

In an exemplary embodiment, a function can be modified by rearranging the bits of the immediate value. The table below shows how rearrangement of immediate value f₇ . . . f₀ can reorder the operands d,c,b for the same function.

operation	immediate
f(d,c,b)	f ₇ f ₆ f ₅ f ₄ f ₃ f ₂ f ₁ f ₀
f(c,d,b)	f ₇ f ₆ f ₃ f ₂ f ₅ f ₄ f ₁ f ₀
f(d,b,c)	f ₇ f ₆ f ₄ f ₃ f ₁ f ₅ f ₂ f ₀
f(c,b,d)	f ₇ f ₆ f ₃ f ₁ f ₄ f ₅ f ₂ f ₀
f(b,d,c)	f ₇ f ₆ f ₂ f ₁ f ₃ f ₄ f ₅ f ₀

In an exemplary embodiment, by using such a rearrangement, an operation of the form: b=f(d,c,b) can be recoded into a legal form: b=f(b,d,c). For example, the function: b=f(d,c,b)=d?c:b cannot be coded, but the equivalent function: d=c?b:d can be determined by rearranging the code for d=f(d,c,b)=d?c:b, which is 11001010, according to the rule for f(d,c,b) f(c,b,d), to the code 11011000.

Encoding

In an exemplary embodiment, some special characteristics of this rearrangement is the basis of the manner in which the eight function specification bits are compressed to seven immediate bits in this instruction. As seen in the table above, in the general case, a rearrangement of operands from f(d,c,b) to f(d,b,c) (interchanging rc and rb) requires interchanging the values of f₆ and f₅ and the values of f₂ and f₁.

In an exemplary embodiment, among the 256 possible functions which this instruction can perform, one quarter of them (64 functions) are unchanged by this rearrangement. These functions have the property that f₆=f₅ and f₂=f₁. The values of rc and rb (Note that rc and rb are the register specifiers, not the register contents) can be freely

US 6,725,356 B2

37

interchanged, and so are sorted into rising or falling order to indicate the value of f_2 . (A special case arises when $rc=rb$, so the sorting of rc and rb cannot convey information. However, as only the values f_7, f_4, f_3 , and f_0 can ever result in this case, f_6, f_5, f_2 , and f_1 need not be coded for this case, so no special handling is required.) These functions are encoded by the values of f_7, f_6, f_4, f_3 , and f_0 in the immediate field and f_2 by whether $rc>rb$, thus using 32 immediate values for 64 functions.

In an exemplary embodiment, another quarter of the functions have $f_6=1$ and $f_5=0$. These functions are recoded by interchanging rc and rb , f_6 and f_5 , f_2 and f_1 . They then share the same encoding as the quarter of the functions where $f_6=0$ and $f_5=1$, and are encoded by the values of f_7, f_4, f_3, f_2, f_1 , and f_0 in the immediate field, thus using 64 immediate values for 128 functions.

In an exemplary embodiment, the remaining quarter of the functions have $f_6=f_5$ and $f_2=f_1$. The half of these in which $f_2=1$ and $f_1=0$ are recoded by interchanging rc and rb , f_6 and f_5 , f_2 and f_1 . They then share the same encoding as the eighth of the functions where $f_2=0$ and $f_1=1$, and are encoded by the values of f_7, f_6, f_4, f_3 , and f_0 in the immediate field, thus using 32 immediate values for 64 functions.

In an exemplary embodiment, the function encoding is summarized by the table:

f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0	$rc > rb$	ih	il_5	il_4	il_3	il_2	il_1	il_0	rc	rb
		f_6				f_2		f_1	0	0	f_6	f_7	f_4	f_3	f_0	rc	rb
		f_6				f_2		$\neg f_2$	0	0	f_6	f_7	f_4	f_3	f_0	rb	rc
		f_6			0	1			0	1	f_6	f_7	f_4	f_3	f_0	rc	rb
		f_6			1	0			0	1	f_6	f_7	f_4	f_3	f_0	rb	rc
0		1							1	f_2	f_1	f_7	f_4	f_3	f_0	rc	rb
1	0								1	f_1	f_2	f_7	f_4	f_3	f_0	rb	rc

In an exemplary embodiment, the function decoding is summarized by the table:

ih	il_5	il_4	il_3	il_2	il_1	il_0	$rc > rb$	f_7	f_6	f_5	f_4	f_3	f_2	f_1	f_0
0	0						0	il_5	il_4	il_3	il_2	il_1	0	0	il_0
0	0						1	il_5	il_4	il_3	il_2	il_1	1	1	il_0
0	1							il_5	il_4	il_3	il_2	il_1	0	1	il_0
1								il_5	0	1	il_2	il_1	il_2	il_1	il_0

From the foregoing discussion, it can be appreciated that an exemplary embodiment of a compiler or assembler producing the encoded instruction performs the steps above to encode the instruction, comparing the f_6 and f_5 values and the f_2 and f_1 values of the immediate field to determine which one of several means of encoding the immediate field is to be employed, and that the placement of the rb and rc register specifiers into the encoded instruction depends on the values of f_2 (or f_1) and f_6 (or f_5).

An exemplary embodiment of the pseudocode 2330 of the Group Boolean instruction is shown in FIG. 23C. It can be appreciated from the code that an exemplary embodiment of a circuit that decodes this instruction produces the f_2 and f_1 values, when the immediate bits ih and il_5 are zero, by an arithmetic comparison of the register specifiers rc and rb , producing a one (1) value for f_2 and f_1 when $rc>rb$. In an

38

exemplary embodiment, there are no exceptions for the Group Boolean instruction.

Improving the Branch Prediction of Simple Repetitive Loops of Code

In yet a further aspect to the present invention, a system and method is described for improving the branch prediction of simple repetitive loops of code. In such a simple loop, the end of the loop is indicated by a conditional branch backward to the beginning of the loop. The condition branch of such a loop is taken for each iteration of the loop except the final iteration, when it is not taken. Prior art branch prediction systems have employed finite state machine operations to attempt to properly predict a majority of such conditional branches, but without specific information as to the number of times the loop iterates, will make an error in prediction when the loop terminates.

The system and method of the present invention includes providing a count field for indicating how many times a branch is likely to be taken before it is not taken, which enhances the ability to properly predict both the initial and final branches of simple loops when a compiler can determine the number of iterations that the loop will be performed. This improves performance by avoiding misprediction of the branch at the end of a loop when the loop

terminates and instruction execution is to continue beyond the loop, as occurs in prior art branch prediction hardware.

Branch Hint

An exemplary embodiment of the Branch Hint instruction is shown in FIGS. 24A–24C. In an exemplary embodiment, this operation indicates a future branch location specified by a register.

In an exemplary embodiment, this instruction directs the instruction fetch unit of the processor that a branch is likely to occur count times at simm instructions following the current successor instruction to the address specified by the contents of register rd . An exemplary embodiment of the format 2410 of the Branch Hint instruction is shown in FIG. 24A.

In an exemplary embodiment, after branching count times, the instruction fetch unit presumes that the branch at simm instructions following the current successor instruction is not likely to occur. If count is zero, this hint directs

US 6,725,356 B2

39

the instruction fetch unit that the branch is likely to occur more than 63 times.

In an exemplary embodiment, an Access disallowed exception occurs if the contents of register rd is not aligned on a quadlet boundary.

An exemplary embodiment of the pseudocode 2430 of the Branch Hint instruction is shown in FIG. 24B. An exemplary embodiment of the exceptions 2460 of the Branch Hint instruction is shown in FIG. 24C.

Incorporating Floating Point Information Into Processor Instructions

In a still further aspect of the present invention, a technique is provided for incorporating floating point information into processor instructions. In related U.S. Pat. No. 5,812,439, a system and method are described for incorporating control of rounding and exceptions for floating-point instructions into the instruction itself. The present invention extends this invention to include separate instructions in which rounding is specified, but default handling of exceptions is also specified, for a particular class of floating-point instructions.

Ensemble Sink Floating-point

In an exemplary embodiment, a Ensemble Sink Floating-point instruction, which converts floating-point values to integral values, is available with control in the instruction that include all previously specified combinations (default-near rounding and default exceptions, Z—round-toward-zero and trap on exceptions, N—round to nearest and trap on exceptions, F—floor rounding (toward minus infinity) and trap on exceptions, C—ceiling rounding (toward plus infinity) and trap on exceptions, and X—trap on inexact and other exceptions), as well as three new combinations (Z.D—round toward zero and default exception handling, F.D—floor rounding and default exception handling, and C.D—ceiling rounding and default exception handling). (The other combinations: N.D is equivalent to the default, and X.D—trap on inexact but default handling for other exceptions is possible but not particularly valuable).

An exemplary embodiment of the Ensemble Sink Floating-point instruction is shown in FIGS. 25A–25C. In an exemplary embodiment, these operations take one value from a register, perform a group of floating-point arithmetic conversions to integer on partitions of bits in the operands, and place the concatenated results in a register. An exemplary embodiment of the operation codes, selection, and format 2510 of Ensemble Sink Floating-point instruction is shown in FIG. 25A.

In an exemplary embodiment, the contents of register rc is partitioned into floating-point operands of the precision specified and converted to integer values. The results are catenated and placed in register rd.

In an exemplary embodiment, the operation is rounded using the specified rounding option or using round-to-nearest if not specified. If a rounding option is specified, unless default exception handling is specified, the operation raises a floating-point exception if a floating-point invalid operation, divide by zero, overflow, or underflow occurs, or when specified, if the result is inexact. If a rounding option is not specified or if default exception handling is specified, floating-point exceptions are not raised, and are handled according to the default rules of IEEE 754.

An exemplary embodiment of the pseudocode 2530 of the Ensemble Sink Floating-point instruction is shown in FIG.

40

25B. An exemplary embodiment of the exceptions 2560 of the Ensemble Sink Floating-point instruction is shown in FIG. 25C.

5 An exemplary embodiment of the pseudocode 2570 of the Floating-point instructions is shown in FIG. 25D.

Crossbar Compress, Expand, Rotate, and Shift

In one embodiment of the invention, crossbar switch units such as units 142 and 148 perform data handling operations, as previously discussed. As shown in FIG. 32A, such data handling operations may include various examples of Crossbar Compress, Crossbar Expand, Crossbar Rotate, and Crossbar Shift operations. FIGS. 32B and 32C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Crossbar Compress, Crossbar Rotate, Crossbar Expand, and Crossbar Shift instructions. As shown in FIGS. 32B and 32C, in this exemplary embodiment, the contents of register rc are partitioned into groups of operands of the size specified, and compressed, expanded, rotated or shifted by an amount specified by a portion of the contents of register rb, yielding a group of results. The group of results is catenated and placed in register rd.

25 Various Group Compress operations may convert groups of operands from higher precision data to lower precision data. An arbitrary half-sized sub-field of each bit field can be selected to appear in the result. For example, FIG. 32D shows an X.COMPRESS rd=rc,16,4 operation, which performs a selection of bits 19 . . . 4 of each quadlet in a hexlet. Various Group Shift operations may allow shifting of groups of operands by a specified number of bits, in a specified direction, such as shift right or shift left. As can be seen in FIG. 32C, certain Group Shift Left instructions may also involve clearing (to zero) empty low order bits associated with the shift, for each operand. Certain Group Shift Right instructions may involve clearing (to zero) empty high order bits associated with the shift, for each operand. Further, certain Group Shift Right instructions may involve filling empty high order bits associated with the shift with copies of the sign bit, for each operand.

Extract

45 In one embodiment of the invention, data handling operations may also include a Crossbar Extract instruction. FIGS. 33A and 33B illustrate an exemplary embodiment of a format and operation codes that can be used to perform the Crossbar Extract instruction. As shown in FIGS. 33A and 33B, in this exemplary embodiment, the contents of registers rd, rc, and rb are fetched. The specified operation is performed on these operands. The result is placed into register ra.

50 The Crossbar Extract instruction allows bits to be extracted from different operands in various ways. Specifically, bits 31 . . . 0 of the contents of register rb specifies several parameters which control the manner in which data is extracted, and for certain operations, the manner in which the operation is performed. The position of the control fields allows for the source position to be added to a fixed control value for dynamic computation, and allows for the lower 16 bits of the control field to be set for some of the simpler extract cases by a single GCOPY1.128 instruction (see appendix). The control fields are further arranged so that if only the low order 8 bits are non-zero, a 128-bit extraction with truncation and no rounding is performed.

US 6,725,356 B2

41

31	24	23	16	15	14	13	12	11	10	9	8	0
fsize		dpos		x	s	n	m	l	rd	gssp		
8		8		1	1	1	1	1	2	9		

The table below describes the meaning of each label:

label	bits	meaning
fsize	8	field size
dpos	8	destination position
x	1	reserved
s	1	signed vs. unsigned
n	1	reserved
m	1	merge vs. extract
l	1	reserved
rd	2	reserved
gssp	9	group size and source position

The 9-bit gssp field encodes both the group size, gsize, and source position, spos, according to the formula $gssp = 512 - 4 * gsize + spos$. The group size, gsize, is a power of two in the range 1 . . . 128. The source position, spos, is in the range 0 . . . $(2 * gsize) - 1$.

The values in the s, n, m, l, and rd fields have the following meaning:

values	s	n	m	l	rd
0	unsigned		extract		
1	signed		merge		
2					
3					

As shown in FIG. 33C, for the X.EXTRACT instruction, when $m=0$, the parameters are interpreted to select a fields from the catenated contents of registers rd and rc, extracting values which are catenated and placed in register ra. As shown in FIG. 33D, for a crossbar-merge-extract (X.EXTRACT when $m=1$), the parameters are interpreted to merge a fields from the contents of register rd with the contents of register rc. The results are catenated and placed in register ra.

Shuffle

As shown in FIG. 34A, in one embodiment of the invention, data handling operations may also include various Shuffle instructions, which allow the contents of registers to be partitioned into groups of operands and interleaved in a variety of ways. FIGS. 34B and 34C illustrate an exemplary embodiment of a format and operation codes that can be used to perform the various Shuffle instructions. As shown in FIGS. 34B and 34C, in this exemplary embodiment, one of two operations is performed, depending on whether the rc and rb fields are equal. Also, FIG. 34B and the description below illustrate the format of and relationship of the rd, rc, rb, op, v, w, h, and size fields.

In the present embodiment, if the rc and rb fields are equal, a 128-bit operand is taken from the contents of register rc. Items of size v are divided into w piles and shuffled together, within groups of size bits, according to the value of op. The result is placed in register rd.

Further, if the rc and rb fields are not equal, the contents of registers rc and rb are catenated into a 256-bit operand.

42

Items of size v are divided into w piles and shuffled together, according to the value of op. Depending on the value of h, a sub-field of op, the low 128 bits ($h=0$), or the high 128 bits ($h=1$) of the 256-bit shuffled contents are selected as the result. The result is placed in register rd.

As shown in FIG. 34D, an example of a crossbar 4-way shuffle of bytes within hexlet instruction (X.SHUFFLE.128 $rd=rcb,8,4$) may divide the 128-bit operand into 16 bytes and partitions the bytes 4 ways (indicated by varying shade in the diagram below). The 4 partitions are perfectly shuffled, producing a 128-bit result. As shown in FIG. 33E, an example of a crossbar 4-way shuffle of bytes within trilet instruction (X.SHUFFLE.256 $rd=rc,rb,8,4,0$) may catenate the contents of rc and rb, then divides the 256-bit content into 32 bytes and partitions the bytes 4 ways (indicated by varying shade in the diagram below). The low-order halves of the 4 partitions are perfectly shuffled, producing a 128-bit result.

Changing the last immediate value h to 1 (X.SHUFFLE.256 $rd=rc,rb,8,4,1$) may modify the operation to perform the same function on the high-order halves of the 4 partitions. When rc and rb are equal, the table below shows the value of the op field and associated values for size, v, and w.

op	size	v	w
0	4	1	2
1	8	1	2
2	8	2	2
3	8	1	4
4	16	1	2
5	16	2	2
6	16	4	2
7	16	1	4
8	16	2	4
9	16	1	8
10	32	1	2
11	32	2	2
12	32	4	2
13	32	8	2
14	32	1	4
15	32	2	4
16	32	4	4
17	32	1	8
18	32	2	8
19	32	1	16
20	64	1	2
21	64	2	2
22	64	4	2
23	64	8	2
24	64	16	2
25	64	1	4
26	64	2	4
27	64	4	4
28	64	8	4
29	64	1	8
30	64	2	8
31	64	4	8
32	64	1	16
33	64	2	16
34	64	1	32
35	128	1	2
36	128	2	2
37	128	4	2
38	128	8	2
39	128	16	2
40	128	32	2
41	128	1	4
42	128	2	4
43	128	4	4
44	128	8	4
45	128	16	4

US 6,725,356 B2

43

-continued

op	size	v	w
46	128	1	8
47	128	2	8
48	128	4	8
49	128	8	8
50	128	1	16
51	128	2	16
52	128	4	16
53	128	1	32
54	128	2	32
55	128	1	64

When rc and rb are not equal, the table below shows the value of the op_{4...} field and associated values for size, v, and w: Op₅ is the value of h, which controls whether the low-order or high-order half of each partition is shuffled into the result.

op _{4..0}	size	v	w
0	256	1	2
1	256	2	2
2	256	4	2
3	256	8	2
4	256	16	2
5	256	32	2
6	256	64	2
7	256	1	4
8	256	2	4
9	256	4	4
10	256	8	4
11	256	16	4
12	256	32	4
13	256	1	8
14	256	2	8
15	256	4	8
16	256	8	8
17	256	16	8
18	256	1	16
19	256	2	16
20	256	4	16
21	256	8	16
22	256	1	32
23	256	2	32
24	256	4	32
25	256	1	64
26	256	2	64
27	256	1	128

Conclusion

Having fully described a preferred embodiment of the invention and various alternatives, those skilled in the art will recognize, given the teachings herein, that numerous alternatives and equivalents exist which do not depart from the invention. It is therefore intended that the invention not be limited by the foregoing description, but only by the appended claims.

We claim:

1. In a system having a data path functional unit having a functional unit data path width, a first memory system having a first data path width, and a second memory system having a data path width which is greater than the functional unit data path width and greater than the first data path width, a method comprising:

copying a first memory operand portion from the first memory system to the second memory system, the first memory operand portion having the first data path width; and

44

copying a second memory operand portion from the first memory system to the second memory system, the second memory operand portion having the first data path width and being catenated in the second memory system with the first memory operand portion, thereby forming catenated data.

2. The method of claim 1 further comprising reading at least a portion of the catenated data which is greater in width than the first data path width.

3. The method of claim 2 further comprising specifying a memory specifier from which a plurality of data path widths of data can be read.

4. The method of claim 3 wherein the memory specifier comprises:

- a memory address;
- a memory size; and
- a memory shape.

5. The method of claim 2 further comprising checking the validity of the first memory operand portion and, if valid, permitting a subsequent instruction to access the first memory operand portion.

6. The method of claim 2 further comprising checking the validity of the second memory operand portion and, if valid, permitting a subsequent instruction to access the second memory operand portion.

7. In a system having a data path functional unit having a functional unit data path width, a first memory system having a first data path width, and a second memory system having a data path width which is greater than the functional unit data path width and greater than the first data path width, a method comprising:

copying a first memory operand portion from the first memory system to the second memory system, the first memory operand portion having the first data path width;

copying a second memory operand portion from the first memory system to the second memory system, the second memory operand portion having the first data path width; and

catenating the second memory operand portion in the second memory system with the first memory operand portion, thereby forming catenated data.

8. The method of claim 7 further comprising reading at least a portion of the catenated data which is greater in width than the first data path width.

9. The method of claim 8 further comprising specifying a memory specifier from which a plurality of data path widths of data can be read.

10. The method of claim 9 wherein the memory specifier comprises:

- a memory address;
- a memory size; and
- a memory shape.

11. The method of claim 8 further comprising checking the validity of the first memory operand portion and, if valid, permitting a subsequent instruction to access the first memory operand portion.

12. The method of claim 8 further comprising checking the validity of the second memory operand portion and, if valid, permitting a subsequent instruction to access the second memory operand portion.

13. In a system having a data path functional unit having a functional unit data path width, a first memory system having a first data path width, and a second memory system having a data path width which is greater than the functional unit data path width and greater than the first data path width, a system comprising:

US 6,725,356 B2

45

a first copying module configured to copy a first memory operand portion from the first memory system to the second memory system, the first memory operand portion having the first data path width; and

a second copying module configured to copy a second memory operand portion from the first memory system to the second memory system, the second memory operand portion having the first data path width and being catenated in the second memory system with the first memory operand portion, thereby forming catenated data.

14. The system of claim 13 further comprising a reading module configured to read at least a portion of the catenated data which is greater in width than the first data path width.

15. In a system having a data path functional unit having a functional unit data path width, a first memory system having a first data path width, and a second memory system having a data path width which is greater than the functional unit data path width and greater than the first data path width, a system comprising:

a first copying module configured to copy a first memory operand portion from the first memory system to the second memory system, the first memory operand portion having the first data path width; and

a second copying module configured to copy a second memory operand portion from the first memory system to the second memory system, the second memory operand portion having the first data path width.

16. The system of claim 15 further comprising a catenating module configured to catenate in the second memory system the second memory operand portion with the first memory operand portion, thereby forming catenated data.

17. The system of claim 16 further comprising a reading module configured to read at least a portion of the catenated data which is greater in width than the first data path width.

18. A method of processing a data stream in a general purpose processor capable of operation independent of another host processor, the general purpose processor having a virtual memory addressing unit, an instruction path and a data path to digitally process the data stream, the method comprising:

receiving the data stream over the data path;

dynamically partitioning the data stream based on an elemental width of the data and storing partitioned data in registers of a register file coupled to the data path, wherein a number of data elements stored in a register is inversely related to the elemental width of the data stored in partitioned fields of the register;

performing group floating point operations on multiple operands stored in partitioned fields of registers and, for each group floating point operation, returning catenated results of the operation to a register.

19. The method of claim 18 wherein the data stream comprises media data.

20. The method of claim 19 wherein the media data comprises broadband communications data.

21. The method of claim 19 wherein the media data comprises audio data.

22. The method of claim 19 wherein the media data comprises image data.

23. The method of claim 19 wherein the media data comprises video data.

24. The method of claim 19 wherein the media data comprises compressed data.

25. The method of claim 19 wherein the media data comprises error checking data.

46

26. The method of claim 19 wherein the media data comprises error correction data.

27. The method of claim 18 wherein, for a specific group floating point operation performed, the catenated results of the specific operation are returned to a register that is different than the registers used to store the multiple operands for the specific operation.

28. The method of claim 18 wherein the performing step comprises performing group add, group subtract and group multiply arithmetic operations on catenated floating-point data and, for each such group operation, returning catenated results of the operation to a register.

29. The method of claim 18 wherein the performing group floating-point operations comprises operating, in parallel, on multiple operands stored in partitioned fields of registers.

30. The method of claim 18 wherein the performing group floating-point operations comprises performing a first group floating-point operation on floating-point data of a first precision and performing a second group floating-point operation on floating-point data of a second precision that is a lower precision than the first precision by operating, in parallel, on at least two operands stored in partitioned fields of registers.

31. The method of claim 18 further comprising performing group integer operations on multiple operands stored in partitioned fields of registers and, for each group integer operation, returning catenated results of the operation to a register.

32. The method of claim 31 wherein, for a specific group integer operation performed, the catenated results of the specific operation are returned to a register that is different than the registers used to store the multiple operands for the specific operation.

33. The method of claim 31 wherein the performing group integer operations comprises performing group add, group subtract and group multiply arithmetic operations on catenated integer data and, for each such group operation, returning catenated results of the operation to a register.

34. The method of claim 31 wherein the performing group integer operations comprises operating, in parallel, on multiple operands stored in partitioned fields of registers.

35. The method of claim 31 wherein the performing group integer operations comprises performing a first group integer operation on integer data of a first precision and performing a second group integer operation on integer data of a second precision that is a lower precision than the first precision by operating, in parallel, on at least two operands stored in partitioned fields of registers.

36. The method of claim 18 further comprising performing one or more group data handling operations that operate on multiple operands stored in partitioned fields of operand registers and, for each group data handling operation, returning catenated results of the operation to a register.

37. The method of claim 36 wherein the performing one or more group data handling operations comprises converting a plurality of n-bit data elements in a first operand register and a plurality of n-bit data elements in a second operand register into a plurality of n/2-bit data elements.

38. The media processor of claim 37 wherein the converting step shifts each of the plurality of n/2-bit data elements by a specified number of bits during the conversion.

39. The method of claim 36 wherein the performing one or more group data handling operations comprises interleaving a plurality of data elements selected from a first operand register with a plurality of data elements selected from a second operand register and catenating the data elements into a result register.

US 6,725,356 B2

47

40. The method of claim 36 wherein the performing one or more data handling operations comprises shifting bits of individual data elements catenated in an operand register to the left and clearing empty low order bits of the individual data elements to zero.

41. The method of claim 36 wherein the performing one or more data handling operations comprises shifting bits of individual data elements catenated in an operand register to the right and filling empty high order bits of the individual data elements with a value equal to a value stored in a sign bit of the individual data element.

42. The method of claim 36 wherein the performing one or more data handling operations comprises shifting bits of individual data elements catenated in an operand register to the right and clearing empty high order bits of the individual data elements to zero.

43. The method of claim 36 wherein the performing one or more group data handling operations comprises operating, in parallel, on multiple operands stored in partitioned fields of registers.

44. The method of claim 36 wherein the performing one or more group data handling operations comprises perform-

48

ing a first group data handling operation on data of a first precision and performing a second group data handling operation on data of a second precision that is a lower precision than the first precision by operating, in parallel, on at least two operands stored in partitioned fields of registers.

45. The method of claim 18 wherein the group floating point operations are associated with a plurality of instruction streams from a plurality of threads executing in parallel on the processor.

46. The method of claim 18 wherein the performing step comprises performing group floating-point operations on data having a total aggregate width of 128 bits.

47. The method of claim 18 wherein the performing step comprises performing group floating-point operations on data of more than one precision.

48. The method of claim 18 further comprising storing floating-point data in a register file in a format conforming to IEEE standard 754.

* * * * *